AN ABSTRACT OF THE THESIS OF

STEPHEN ANTHONY DUM for the degree DOCTOR OF PHILOSOPHY
(Name) (Degree)

in Computer Science presented on March 21, 1975
(Major Department) (Date)

Title: A FAST DIVISION ALGORITHM

Abstract approved: Redacted for privacy
Professor Harry E. Goheen

A radix 2^n non-restoring division algorithm is described. The algorithm is designed to be compatible with hardware multiprecision multiplication methods currectly used in high speed digital computers. This enables the use of the same hardware, with only changes in control logic, to be used to implement both multiplication and division. This paper proves that in order to obtain n bits of the quotient at each iteration in a non-restoring algorithm it is only necessary to consider the first n + 3 bits (including the sign as one bit) of the divisor and the dividend to obtain a quotient estimator.

A section is devoted to implementation of the algorithm in software as a way to extend the precision of the existing hardware division instruction on a digital computer.

A Fast Division Algorithm

by

Stephen Anthony Dum

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

March 21, 1975

Commencement June 1975

APPROVED:

Redacted for privacy

Professor of Computer Science

Redacted for privacy

Chairman of Department of Computer Science

Redacted for privacy

Dean of Graduate School

Date thesis is presented March 21, 1975

Typed by Linda Dum for Stephen Anthony Dum

TABLE OF CONTENTS

I. Background..1

II. Higher Radix Division.............................9

III. Definition of the Algorithm.......................16

IV. An Analysis of the Algorithm......................22

V. Hardware Implementation...........................43

VI. Implementation of the Division Procedure in Software...56

VII. Summary..61

Bibliography.....................................64

Appendix A: Index to Defined Symbols.................67

Appendix B: Definition of Abbreviations..............69

LIST OF ILLUSTRATIONS

Figure		Page
1	Adding six numbers with a carry save adder tree.	50
2	A six bit division example.	52

LIST OF TABLES

Table		Page
1	Maximum number of bits developed for p and m.	35
2	Values of n, given p and m fixed with a negative dividend.	42

A Fast Division Algorithm

Background

The mechanization of arithmetic operations on a binary digital computer has gone through many stages, trying to minimize both the cost of implementation and the execution time. Our concern here is with the division, but much of the work on multiplication is analogous, just as addition is to subtraction, since they are inverse operations. The first implementations of these operations [32] were implementations of the methods used in similar hand calculations. These algorithms are of the shift and subtract type, proceeding one digit at a time, calculating the quotient.

For the sake of clarity, let us assume that both the divisor and the dividend are positive. We will describe the more general case, that of generating a quotient digit in an arbitrary but specified radix, b.

One subtracts the divisor from the dividend (at later stages, an adjusted form of the partial remainders) until the result is less than the divisor. This will occur after i subtractions, $0 \leq i \leq b-1$. The quotient digit generated by this process is i and the partial remainder is multiplied by b to become the new dividend. This multiplication is trivial if the numbers are represented in radix b (or if b is a power of the radix the numbers are represented in), as it is simply a shift operation. A comparison is required for all values of i except for the last, $i = b - 1$. If we reach the

situation that $i = b - 1$ in a well-conducted division no comparison is necessary, since each quotient digit must be less than b.

The complexity of a comparison is equivalent to that of an addition or a subtraction. Thus the above process requires the equivalent of two subtractions of each value of i. Two methods which reduce the number of operations required are the restoring and non-restoring methods of division.

In the restoring method of division one repeatedly subtracts the divisor from the dividend until the results becomes negative (remembering our assumption that both the divisor and the dividend are initially positive), counting the number of subtractions as before. When the result becomes negative we add the divisor to the result, and decrement our counter, i. The correct quotient digit is i. The partial remainder multiplied by b, becomes the dividend for the next iteration and we repeat this process.

The restoring method eliminates all the comparisons, introducing one correcting operation at the end of each cycle. The non-restoring method is an attempt to eliminate the one correction, still remaining in the restoring method. This is important for small radices, especially two. In radix two, if we assume a random distribution of quotient digits, half of the time the quotient digit is zero, necessitating a correction. This means that, for radix two, the restoring method is no improvement over the subtract and compare method.

In the non-restoring method, the addition at the end of each

cycle is eliminated. Again we assume initially that both the divisor and the dividend are positive. We proceed as follows: use i to count the number of iterations in each cycle. If the dividend is positive, we subtract the divisor from the dividend. When the dividend changes sign the cycle is complete and i is the quotient digit. If the dividend is negative we add the divisor to the dividend. When the dividend changes signs the cycle is complete and -i is the quotient digit. In other words, the quotient digit is -i times the sign of the dividend. Thus the quotient digits take on values of ± 1, $\pm 2, \ldots \pm (b-1)$, $\pm b$, or for a binary computer ± 1, ± 2.

Most computers do not allow for the redundant representation obtained here. This would require 2b digits instead of the usual b digits [3], [4], [5], [14], and [28].

More analysis of the matter shows there is an obvious and simple relation among the digits of the quotient. The first digit of the quotient is always positive and the sign of each quotient digit alternate thereafter. Also zero is not one of the valid digits. This relationship is best illustrated by an example. If two digits were 5 and -3 the conversion required would be to borrow one from the 5 and add b to -3, obtaining 4 and b-3. In general, we can incorporate this processing into the non-restoring algorithm. One way to do this is: If the dividend is positive, start counting with $i = 0$ rather than one. When the dividend changes signs i is the quotient digit. If the dividend is negative, start counting with $i = b - 1$. Here we decrement i at each iteration. When the

dividend changes sign i is the correct quotient digit. After the nth iteration i will have the value of b - n. This process or a similar one avoids the redundant representation of the quotient but uses the non-restoring algorithm.

If we reduce the non-restoring algorithm to radix two the process simplifies greatly. The only possible quotient digits are zero and one, thus the counter i is not needed. We perform one subtraction or addition and the sign of the dividend along with the knowledge of whether we added or subtracted determine the quotient digit. Further information on these methods as related to binary computers can be found in Flores [12] or Chu [8].

If we restrict the values of the divisor, we can increase the speed of the previous discussed methods. If we assume the first digit of the divisor is non-zero, when the dividend is positive, a string of zeros at the start of the dividend would indicate that the next quotient digit is zero. Thus, in some cases we can determine the next quotient digit without any additions or subtractions. Mac Sorley [21] gives a complete discussion of this in the binary case. He shows that for very sophisticated algorithms, an addition or subtraction is needed only one out of every 3.8 cycles on the average.

The development of multiplication followed that of division as we have described it. However, the process of multiplication was further speeded up by performing more than one digit at a time [29]; forming the product of the multiplicand and several digits of the

multiplier (rather than just one digit of the multiplier) at each iteration. There has been no parallel development for division. In division, each successive digit depends on the results of the calculations determining the last quotient digit. Thus, the development of division has tended to use multiplication as the basic iterative tool, taking advantage of the high speed algorithms for multiplication [1], [6], [11], and [13].

The first class of algorithms for division using multiplication as the basic tool were formulated around a method proposed by Newton, to approximate a zero of an arbitrary function, viz.,

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} .$$

In applying this to the division problem, Q = A/B one first calculates 1/B and then multiplies this result by A. Using Newton's formula with $f(x) = \frac{1}{x} - B$ we obtain

$$x_{n+1} = 2x_n - Bx_n^2. \tag{1}$$

This method converges quadratically to 1/B. In other words, if $x_n = 1/B + \varepsilon$ then $x_{n+1} = 1/B - B\varepsilon^2$. To use this method effectively one must provide as accurate an estimate of 1/B as possible. This assures that the algorithm converges rapidly to 1/B.

The assumption that the divisor is normalized is almost essential using this method. If we know that the divisor is in the range $1 > B \geq 1/2$ then $2 \geq 1/B > 1$, otherwise the range of 1/B

can become unmanageable.

A very simple approach is to approximate 1/B with 1. That is, use $x_0 = 1$ as the starting value to find 1/B. If this is done the relative error is $|\varepsilon| \leq 1/2$. Gill suggested in 1955, [15] using $x_0 = 4(\sqrt{3}-1) - 2B$ for the starting value. Since the first term is a constant the calculation involves one subtraction and one shift. Using this the relative error is $|\varepsilon| \leq 2^{-3.75}$. In 1968, Dean [10] suggested what appears to be a much simplier equation, viz., $x_0 = 3 - 2B$, which gives $|\varepsilon| \leq 2^{-3}$. This equation requires just as much computing as the method of Gill as both 3 and $4(\sqrt{3}-1)$ are constants. Another approach is to examine the first k bits of the dividend and use these in determining the first approximation of the reciprocal of the quotient. Wallace [33] examined the first six bits and obtained the relative error $|\varepsilon| \leq 2^{-5}$.

Using Newton's iterative approximation for the reciprocal of the dividend requires two dependent multiplications at each iteration. A method used by Anderson [2], known as the Harvard iteration scheme [25] speeds up this process. In this method, there are still two multiplications required at each iteration but they are independent of each other. At each iteration both the divisor and the dividend tend toward one. In calculating A/B we would have

$$\frac{A*G_0*G_1*\cdots*G_n}{B*G_0*G_1*\cdots*G_n} \sim A*G_0*G_1*\cdots*G_n = Q$$

since $B*G_0*G_1*\cdots*G_n \sim 1$.

Anderson [2] determined G_0 by table lookup using the first seven bits of the dividend, claiming an error

$$|\varepsilon| \leq 2^{-7}.$$

Determination of G_k can be found as follows. We consider $B*G_0*\cdots*G_{k-1}$ to be of the form $1-\varepsilon$ and let G_k be $1+\varepsilon$. Then $B*G_0*\cdots*G_k = 1-\varepsilon^2$. Thus the dividend, $BG_0*G_1*\cdots*G_k$ is converging quadratically to one, while the divisor is converging to the quotient.

The calculations of G_k is then simply,

$$G_k = 2 - B*G_0*G_1*\cdots*G_{k-1}.$$

For the fractional numbers this operation is simply the two's complement of the number.

In the methods described using multiplication as the iterative tool, we have the obvious restriction that the speed of the division is slower than that of a multiplication. The method proposed in this paper does not use multiplication in the iterative process but rather uses the same methods as used in the fast multiplication scheme. This enables one to obtain equivalent execution times for both multiplication and division.

The fastest and probably the most expensive method of division is to perform the entire operation in parallel. This can be done by a table lookup or a network of logic that obtains the results directly [7]. In cases where the operands are reasonably large (perhaps more than eight bits by today's technology) these methods

are simply cost prohibitive.

Multiplication speed has been increased from the single bit at a time iterative procedures by using higher radix or multiprecision methods. In these methods n bits of the multiplier are multiplied by the multiplicand at each iteration. Ideally we would like to have an algorithm for multiplication and one for division which involve essentially the same operations at each iteration, so that the same hardware could be used for both with only some changes in control logic. So far, methods proposed have failed to do this. Rather they have used high speed multiplication as the basic iterative tool.

While there have been some higher radix division methods proposed, they do not provide the ease of implementation that the methods of multiplication do, because of constraints and peculiarities involved in their implementation. The method proposed here, on the contrary avoids these problems and can be implemented in the same hardware as for multiplication,changing only some control logic.

Higher Radix Division

In the restoring and non-restoring methods, the approach is to subtract (or in some cases add) the divisor to the dividend until the sign changes, thus determining the number of times the divisor goes into the dividend. Most implementations on binary computers have used radix two. This simplifies the calculations as the quotient bit is either one or zero.

In order to obtain speed comparable to multiplication we need to obtain more than one bit of the quotient at each iteration. This can be done by using radix 2^n, and rather than using the repetitive subtraction approach, to obtain an estimate of the correct quotient digit and use that. The problem is that if our estimate of the quotient digit is not correct we still have to make some corrections. Stein and Pope [31] proposed a method of estimating the quotient which required up to four tries to get the correct quotient. This was improved by Stein [30] to three tries. Mifsud [22] used the first two digits of the dividend and one digit of the divisor (in radix b). He was able to obtain the correct quotient in two tries in most cases, but required some sophisticated checks and possible adjustments to precondition the divisor before starting. Flores [12] mentions the concepts of a radix 2^n division algorithm. He describes the development of two bits of the quotient looking at 3, 4 and 6 bits of both the dividend and the divisor. He mentions the development of three bits of the quotient but suggests that the efficiency

of this is so low as to warrant no further consideration, much less consider development of more than three bits at a time.

The method proposed here eliminates the pre-operative conditioning of the divisor, requiring only that it be in normalized form. That is, the divisor is in the interval [1/2, 1). Since the implementation is directed toward a binary computer, only 2^n is considered as a radix for the operations. We assume that the divisor and the dividend are initially positive. The dividend is replaced by the partial remainder during the process and the partial remainder may be negative.

The method obtains an estimate c, of the quotient digit C, in such a manner that the correct quotient digit is either c or c-1. Using c as the trial quotient digit, if when subtracting c times the divisor from the dividend, the result is negative then the actual quotient digit is c-1, otherwise it is c. We have the same decisions as to the method of implementation as before, that is, either restoring or non-restoring. The non-restoring method turns out to be as simple as the restoring method, and it is faster. The restoring method requires the addition of the divisor back into the dividend whenever the dividend becomes negative after subtracting c times the divisor from it. This correction will be sufficient, because c was either correct or equal to C+1.

In the non-restoring case we must modify our estimate c of C when the dividend is negative to compensate for the incorrect

estimate of the preceding step of the iteration.

In the following discussion we will assume that the dividend A is less than the divisor B and that the binary points of both the divisor and the dividend are located after the sign digit. In other words initially we can represent A and B as follows:

$$A = 0.x\cdots x$$

$$B = 0.x\cdots x$$

where x represents an arbitrary bit (zero or one). We assume the quotient digit C and its estimate c, are positive integers in the range 0, 1, ..., 2^{n-1}. These assumptions do not affect the generality of the method but do aid in the clarification of some of the numerical calculations.

Let us first consider the case when the dividend is positive. The partial remainder A', is calculated by

$$A' = A - c*B*2^{-n}. \tag{2}$$

The dividend for the next iteration is

$$A = A'*2^{n} \tag{3}$$

where n is the number of bits of the quotient obtained at each iteration.

The actual quotient digit C is determined by

$$C = \left[\frac{A}{B}\, 2^{n}\right]_{GI}. \tag{4}$$

In order to obtain an estimate c of C which is either C or C+1, we use estimates of the divisor and the dividend. These

estimates will be formed by looking at truncated forms of A and B. It is convenient if these truncated forms are integral. They will be multiplied by appropriate powers of two to obtain this. In the case of A, our estimate a, will be formed by looking at the first p bits of A. The actual value of p will be determined later and should be as small as possible and still have our estimate of c lie between C and C+1. Thus a is calculated by:

$$a = [A*2^{p-1}]_{GI}. \tag{5}$$

We can write a as follows:

$$a = \underbrace{x\cdots x}_{p}.$$

where the first bit, the sign bit is initially zero, but if the dividend becomes negative, it becomes a one. Since we must examine at least the sign of the dividend at each iteration, p must be greater than zero.

The divisor is represented similarly. We know that the divisor is always positive and it is normalized. Thus $B = 0.1x\cdots x$. Our estimate b of B is determined by looking at the first m bits of B. This includes the sign and the next bit of B, which are always zero and one, respectively. The value of m, just as the value of p, will be determined later. In fact, the problem of determining p and m are interrelated. The accuracy of the estimate c of C is dependent on both p and m. The larger we choose p and m the more accurate our estimate c.

The estimate b is calculated by:

$$b = [B*2^{m-1}]_{GI}. \tag{6}$$

Thus we can represent b in binary as

$$b = \underbrace{01x\cdots x}_{m}.$$

Since the first two bits of the divisor are always zero and one we have $m \geq 2$.

With this groundwork we can return to the estimate C when the dividend is positive. From the definition of a and b (equations 5 and 6) we can write

$$A*2^{p-1} \in [a, a+1), \tag{7}$$

$$B*2^{m-1} \in [b, b+1). \tag{8}$$

Equation 4 expressed the value of C in terms of A and B, viz.,

$$C = \left[\frac{A}{B} 2^n\right]_{GI}.$$

From 7 and 8 we see that

$$\frac{a}{b+1}t \leq C < \frac{a+1}{b}t,$$

where $t = 2^{n+m-p}$. By looking at our estimate a and b of A and B we can assert that C (an integer) is in the closed interval

$$C \in \left[\left[\frac{a}{b+1} t\right]_{GI}, \left[\frac{a+1}{b} t\right]_{GI}\right]. \tag{9}$$

Since we have a well formed division problem we can make the

interval smaller than the interval specified in 9. We know that C is less than 2^n. Thus, C is in the closed interval

$$C \in \left[\left\lceil \frac{a}{b+1} t \right\rceil_{GI}, \min\left(\left\lceil \frac{\bar{a}+1}{b} t \right\rceil_{GIL}, 2^n-1 \right) \right]. \qquad (10)$$

Since our estimate c of C must be such that either c = C or c = C+1 we choose for our estimate of C the largest value of C in the interval, viz.,

$$c = \min\left(\left\lceil \frac{\bar{a}+1}{b} t \right\rceil_{GIL}, 2^n-1 \right). \qquad (11)$$

Our later determination of p and m will be such that if the quotient digit were actually $C = \left\lceil \frac{a}{b+1} t \right\rceil_{GI}$, then the difference between c and C must not be more than one.

Let us consider the case of a negative dividend A. We must consider how this came about, since initially the dividend is positive. In order for the dividend to be negative the estimate c of C in the last iteration must have been wrong. In other words, since c is either C or C+1, it must have been that c was C+1. Looking at equation 2 we see that this results in $B*2^{-n}$ being subtracted one extra time from the dividend. Equation 3 then multiplies this by 2^n. At this point we have the dividend A, in error by $-2^n*(B*2^{-n})$. This can be corrected by modifying equation 2. We add this amount back in, obtaining

$$A' = A - c*B*2^{-n} + 2^n*B*2^{-n}$$

thus

$$A' = A - (c - 2^n)*B*2^{-n}. \tag{12}$$

In order to calculate the estimate c of the quotient digit, we must take these changes into account. The actual quotient digit is

$$C = \left[\frac{A+B}{B} 2^n\right]_{GI} = 2^n + \left[\frac{A}{B} 2^n\right]_{GI}.$$

Using our estimates a and b we have

$$2^n + \left[\frac{a}{b}t\right]_{GI} \leq C \leq 2^n + \left[\frac{a+1}{b+1}t\right]_{GIL}. \tag{13}$$

Since C is non-negative, the actual value of the quotient lies in the closed interval

$$C \varepsilon \left[\max\left(0,\ 2^n + \left[\frac{a}{b}t\right]_{GI}\right),\ 2^n + \left[\frac{a+1}{b+1}t\right]_{GIL}\right]. \tag{14}$$

As in the positive case, we choose for our estimate c of C the largest value that C can attain in the interval 14,

$$c = 2^n + \left[\frac{a+1}{b+1}t\right]_{GIL}. \tag{15}$$

With the derivation of the quotient estimate we can proceed with the definition of the algorithm.

Definition of the Algorithm

The algorithm to be proposed is a non-restoring division algorithm with each quotient digit calculated as indicated previously. Before formally stating the algorithm, we will consider the assumptions made.

ASSUMPTIONS:

1. We assume the dividend and the divisor are positive. This assumption is reasonable as it requires at most negating the divisor or dividend if they are negative before the division and negating the quotient if necessary afterwards.
2. We assume the divisor is normalized. This assumption is usually met in floating point arithmetic operations and if it is not met it is accomplished simply by shifting the divisor before the division operation (and shifting the quotient appropriately after the operation).
3. We assume that the first bit of each number (most significant or left most bit) is the sign bit; that it is a zero, indicating the numbers are positive; that the binary point is immediately following the sign bit; and that the normal binary place values apply. The three commonly used number systems, one's complement, two's complement and sign-magnitude all satisfy these requirements.
4. We assume that the divisor, quotient and remainder are

represented in r bit numbers (sign and r-1 bits), and that the dividend is of length 2r-1 (the dividend is stored in two r bit words with the sign bit of the second word unused).

5. We assume that the divisor and the dividend satisfy the inequality: dividend < divisor. Failure of this requirement results in the standard "Divide Fault" condition. That is, the quotient is too large to represent. If the inequality were to fail the situation could arise that the quotient is greater than or equal to one. The quotient cannot be represented according to our convention (assumption 3). This is because the divisor is in the half-open interval [1/2,1) and the dividend is in the half-open interval [0,1).

6. We assume the existence of two integers p and m whose value is a function of n, the number of bits of the quotient developed at each stage. These numbers are used in the calculation of approximations of the divisor and the dividend, which in turn are used to calculate the quotient. (Methods for determination of p and m will be described later, in the analysis of the algorithm).

From this list of assumptions we see the only real restriction required by the proposed method is in assumptions 1 and 2. Both of these assumptions can be met by simple and quick operations available on most digital computers.

ALGORITHM A. Radix 2^n, non-restoring division. Given the assump-

tions listed above calculate Q = A/B, where A is the dividend, B is the divisor, and Q is the quotient.

A1. [Initialization] Set $f \leftarrow 2r - n\left\lceil \frac{r}{n} \right\rceil_{GIL} - 2$,

$R \leftarrow \left[A*2^{f}\right]_{GI} *2^{-r-n+1}$, $t \leftarrow 2^{m+n-p}$, $b \leftarrow [B*2^{m-1}]_{GI}$,

$Q \leftarrow 0$, $i \leftarrow 0$, $A \leftarrow \left(A*2^{f} - [A*2^{f}]_{GI} \right)$.

A2. [Calculate a] $a \leftarrow [R*2^{p-1}]_{GI}$.

A3. [Calculate c] If R is positive, set $c \leftarrow \min\left(\left[\frac{a+1}{b}t\right]_{GI}, 2^{n}-1 \right)$,

or else set $c \leftarrow 2^{n} + \left[\frac{a+1}{b+1}t\right]_{GIL}$.

A4. [Calculate partial remainder] Set $R \leftarrow R - c*B*2^{-n} + \begin{Bmatrix} 1 \text{ if } R < 0 \\ 0 \text{ if } R \geq 0 \end{Bmatrix} *B$.

A5. [Adjust c if underflow] If R is negative, set $c \leftarrow c-1$.

A6. [Shift next digit into quotient] Set $Q \leftarrow Q*2^{n} + c*2^{r-1}$.

A7. [Increment loop counter] Set $i \leftarrow i + 1$. If $i = \left[\frac{r}{n}\right]_{GI} + 1$ then A9, otherwise A8.

A8. [Shift next n bits of dividend into R]

$R \leftarrow R*2^{n} + [A*2^{n}]_{GI} * 2^{-n-r+1}$, $A \leftarrow A*2^{n} - [A*2^{n}]_{GI}$. Go to A2.

A9. [Finished. Adjust remainder] Set $R \leftarrow \left(R + \begin{Bmatrix} B \text{ if } R < 0 \\ 0 \text{ if } R > 0 \end{Bmatrix} \right) * 2^{n}$.

Stop (R = remainder $* 2^{r-1}$, q = quotient).

To help clarify this algorithm an example will be given, but first a more intuitive clarification. Most of the operations in the algorithm are shifts of the bit patterns of the numbers. R is an accumulator where the actual division operations occur. In step A1 f bits of the dividend are shifted into R, thereafter at each iteration the next n bits of the dividend are shifted into R. After each iteration the n derived bits of the quotient are shifted into the low order bits of the quotient Q (step A6). Finally when the operation is complete, the remainder is corrected (A9). It is clear that this algorithm stops. The variable i is used as an iterative counter. After $\left[\frac{r}{n}\right]_{GI}$ iterations the algorithm stops, going to step A9.

In the following example all numbers will be represented in two's complement form. The first column lists the step of the algorithm being performed. The following columns list the values of each variable. The example will be for n = 2, m = 3, p = 4, and r = 7. The dividend and the divisor will be A = 0.010100001001 and B = 0.111100.

Example of Division Problem

	f	t	i	a	b	c	R	A	Q
A1	6	2	0000		011		0.00010100	0.001001	0.000000
A2	6	2	0000	0000	011		0.00010100	0.001001	0.000000
A3	6	2	0000	0000	011	000	0.00010100	0.001001	0.000000
A4	6	2	0000	0000	011	000	0.00010100	0.001001	0.000000
A5	6	2	0000	0000	011	000	0.00010100	0.001001	0.000000
A6	6	2	0000	0000	011	000	0.00010100	0.001001	0.000000
A7	6	2	0001	0000	011	000	0.00010100	0.001001	0.000000
A8	6	2	0001	0000	011	000	0.01010000	0.100100	0.000000
A2	6	2	0001	0010	011	000	0.01010000	0.100100	0.000000
A3	6	2	0001	0010	011	010	0.01010000	0.100100	0.000000
A4	6	2	0001	0010	011	010	1.11011000	0.100100	0.000000
A5	6	2	0001	0010	011	001	1.11011000	0.100100	0.000000
A6	6	2	0001	0010	011	001	1.11011000	0.100100	0.000001
A7	6	2	0010	0010	011	001	1.11011000	0.100100	0.000001
A8	6	2	0010	0010	011	001	1.01100010	0.010000	0.000001
A2	6	2	0010	1.011	011	001	1.01100010	0.010000	0.000001
A3	6	2	0010	1.011	011	001	1.01100010	0.010000	0.000001
A4	6	2	0010	1.011	011	001	0.00010110	0.010000	0.000001
A5	6	2	0010	1.011	011	001	0.00010110	0.010000	0.000001
A6	6	2	0010	1.011	011	001	0.00010110	0.010000	0.000101
A7	6	2	0011	1.011	011	001	0.00010110	0.010000	0.000101
A8	6	2	0011	1.011	011	001	0.01011001	0.000000	0.000101

Example of Division Problem
Continued

	f	t	i	a	b	c	R	A	Q
A2	6	2	0011	0.010	011	001	0.01011001	0.000000	0.000101
A3	6	2	0011	0.010	011	010	0.01011001	0.000000	0.000101
A4	6	2	0011	0.010	011	010	1.11100001	0.000000	0.000101
A5	6	2	0011	0.010	011	001	1.11100001	0.000000	0.000101
A6	6	2	0011	0.010	011	001	1.11100001	0.000000	0.010101
A7	6	2	0100	0.010	011	001	0.01110100	0.000000	0.010101
A9	6	2	0100	0.010	011	001	0.01110100	0.000000	0.010101

An Analysis of the Algorithm

Algorithm A relies on the existance of two numbers p and m, which are a function of n, the number of bits of the quotient developed at each iteration. Our estimate c of C must be such that c = C or c = C+1. The error of our estimate is the difference between c and C. Our choice of p and m must be such that this error (C-c) is always less than or equal to one. The determination of p and m follows.

Let us consider the case $A \geq 0$. From (10) we know C is in the interval

$$C \in \left[\left[\frac{a}{b+1}\, t\right]_{GI},\ \min\left(\left[\frac{a+1}{b}\, t\right]_{GIL},\ 2^n - 1\right)\right] .$$

Our estimate c of C is the largest value C might be in this interval, viz.,

$$c = \min\left(\left[\frac{a+1}{b} t\right]_{GIL},\ 2^n - 1\right) .$$

Let C_{min} be the smallest value that C can attain in this interval. The maximum error of our estimate is

$$E = c - C_{min}$$

$$= \min\left(\left[\frac{a+1}{b} t\right]_{GIL},\ 2^n - 1\right) - \left[\frac{a}{b+1}\, t\right]_{GI} . \qquad (16)$$

We will approximate E by $\tilde{E}$. $\tilde{E}$ will be derived from E as follows:

$$E = c - C_{min}$$

$$= \min\left(\left[\frac{a+1}{b}t\right]_{GIL}, 2^n - 1\right) - \left[\frac{a}{b+1}\,t\right]_{GI}$$

$$\leq \left[\frac{a+1}{b}t\right]_{GIL} - \left[\frac{a}{b+1}\,t\right]_{GI}$$

$$\geq \frac{a+1}{b}t - \left[\frac{a}{b+1}\,t\right]_{GI}.$$

Since $[x]_{GI} > x - 1$ we have

$$E < \frac{a+1}{b}t - \frac{a}{b+1}t + 1.$$

Define $\tilde{E}$ as follows:

$$\tilde{E} = \frac{a+1}{b}t - \frac{a}{b+1}\,t = \frac{a+b+1}{b(b+1)}t \tag{17}$$

then we have

$$E < \tilde{E} + 1.$$

If we impose the restriction $\tilde{E} \leq 1$, then we will have $E < 2$ (which is equivalent to $E \leq 1$, since E is an integer).

The maximum value of $\tilde{E}$ can be found by considering equation 17. Since this function is increasing in a and decreasing in b, the maximum value of the function will be on the left boundary, namely on the line $\frac{a}{b}t = 2^n$. In fact, the maximum will be at $a = 2^{p-2}$ and $b = 2^{m-2}$. Evaluating $\tilde{E}$ we find

$$\tilde{E} = \frac{2^{p-2} + 2^{m-2} + 1}{2^{p-2}\,(2^{m-2} + 1)}\, 2^n .$$

Applying the constraint $\tilde{E} \leq 1$ and solving for n we can obtain a lower bound for n, the number of bits of the quotient obtainable as a function of the number of bits of the dividend and the divisor, p and m respectively, that we look at to estimate the quotient.

$$1 \geq \tilde{E} = \frac{2^{p-2} + 2^{m-2} + 1}{2^{p-2}\,(2^{m-2} + 1)}\, 2^n .$$

Solving this inequality for n we obtain

$$2^n \leq \frac{2^{p-2}\,(2^{m-2} + 1)}{2^{p-2} + 2^{m-2} + 1}$$

$$n \leq \log_2 \left(\frac{2^{p-2}\,(2^{m-2} + 1)}{2^{p-2} + 2^{m-2} + 1} \right) . \qquad (18)$$

For any value of n satisfying inequality 18, we are assured that $E < 1$. However, since inequality 18 is derived from an upper bound for the error (equation 17) rather than the error equation itself, it is possible that larger values of n will result in $E < 1$. The value of n:

$$n = \left[\log_2 \left(\frac{2^{p-2}\,(2^{m-2} + 1)}{2^{p-2} + 2^{m-2} + 1} \right) \right]_{GI} \qquad (19)$$

is a lower bound for the number of bits of the quotient we can obtain for a given p and m.

The variables p and m represent the number of bits of the divisor and the dividend that we need to look at to obtain the quotient digit estimate. Since two bits of the divisor are fixed (the sign and the first bit) we need to look at m+p-2 bits of information, or 2^{m+p-2} possibilities. We will show that for any $n > 3$, in order to obtain $E \leq 1$ and the sum of m+p-2 minimum, we must choose

$$p = n + 3$$

and

$$m = n + 3.$$

First, if we let p and m assume these values, then equation 19 is satisfied. The remainder of the proof is by contradiction. If we assume that either p or m is less than n + 3, say by δ or λ respectively, and that the sum m+p-2 does not increase, then we will show that the error E (from equation 16) is greater than two.

Let δ and λ be non-negative integers defined as follows:

$$\delta = n + 3 - p \qquad 0 \le \delta \le n + 2$$

$$\lambda = n + 3 - m \qquad 0 \le \lambda \le n + 1.$$

The range of δ and λ is derived from the fact that $p \ge 1$ and $m \ge 2$. Equation 16 is

$$E = \min\left(\left[\frac{a+1}{b}t\right]_{GIL}, 2^n - 1\right) - \left[\frac{a}{b+1}t\right]_{GI}$$

where

$$t = 2^{m+n-p}$$

$$a \in [0, 2^{p-1})$$

$$b \in [2^{m-2}, 2^{m-1})$$

and since the dividend is less than the divisor we know that

$$a < (b + 1)\, 2^{p-m}.$$

The proof is done by considering cases. First the range of δ will be narrowed by considering the case $a = 0$ and $b = 2^{m-2}$. If we assume $\delta \ge 1$ equation 16 becomes

$$E = \left[\frac{t}{b}\right]_{GIL} = \left[\frac{2^{m+n-p}}{2^{m-2}}\right]_{GIL} = 2^{n-p+2} - 1 = 2^{\delta-1} - 1.$$

The only time $E \le 1$ is when $\delta < 3$. In order to eliminate the other values of δ and λ we will choose as a the smallest value which still makes $\left[\frac{a+1}{b}t\right]_{GIL} \ge 2^n - 1$. This will be $a = \left[\frac{(2^n - 1)b}{t} - 1\right]_{LIG} = \left[\frac{(2^n - 1)b}{t}\right]_{GI}$, since when $a = \frac{(2^n - 1)b}{t} - 1$ we have $\frac{a+1}{b}t =$

$\frac{(2^n - 1)b}{t} \frac{t}{b} = 2^n - 1$. The first term of equation 16 has the value $2^n - 1$ for this choice of a. It is only necessary to evaluate the second term of equation 16 to determine E. If $\frac{a}{b+1}t < 2^n - 2$ then $E > 1$. If we let $b = 2^{n+1-\lambda}$, we can narrow down the range of λ. Evaluating a we obtain

$$a = [2^{n+1-\delta} - 2^{1-\delta}]_{GI}.$$

If $\delta > 1$ we calculate

$$\frac{a}{b+1}t = \frac{2^{n+1-\delta} - 1}{2^{n+1-\lambda} + 1} 2^{n+\delta-\lambda} = \frac{2^n - 2^{\delta-1}}{1 + 2^{\lambda-n-1}} < 2^n - 2.$$

If $\delta \leq 1$, we calculate

$$\frac{a}{b+1}t = \frac{2^{n+1-\delta} - 2^{1-\delta}}{2^{n+1-\lambda} + 1} 2^{n+\delta-\lambda} = \frac{2^n - 1}{1 + 2^{\lambda-n-1}}$$

$$= 2^n - 1 - (2^{\lambda-1} - 2^{\lambda-n-1}) + (2^{2\lambda-n-2} - 2^{2\lambda-2n-2}) - \dots$$

$$< 2^n - 2 \quad \text{whenever} \quad \lambda \geq 2.$$

This shows the error E in equation 16 will be greater than one in all cases where $n > 3$ except when $\delta \leq 1$ and $\lambda \leq 1$. When $\lambda = 1$ and $\delta \leq 1$ let $b = 2^n + 1$. Then a will be

$$a = [2^{n+1-\delta} - 2^{1-\delta-n}]_{GI} = 2^{n+1-\delta} - 1.$$

We can calculate

$$\frac{a}{b+1}t = \frac{2^{n+1-\delta} - 1}{2^n + 2} 2^{n+\delta-1} = \frac{2^n - 2^{\delta-1}}{1 + 2^{1-n}}$$

$$= 2^n - 2^{\delta-1} - (2 - 2^{\delta-n}) + (2^{2-n} - 2^{\delta+1-2n}) - \ldots$$

$$< 2^n - 2.$$

Finally, when $\lambda = 0$ and $\delta = 1$ let $b = 2^{n+1} + 2$. Then a will be

$$a = [2^n - 2^{-n}]_{GI} = 2^n - 1.$$

We calculate

$$\frac{a}{b+1}t = \frac{2^n - 1}{2^{n+1} + 3} 2^{n+1} = \frac{2^n - 1}{1 + 3 \cdot 2^{-1-n}}$$

$$= 2^n - 1 - (3 \cdot 2^{-1} - 3 \cdot 2^{-1-n}) + (9 \cdot 2^{-2-n} - 9 \cdot 2^{-2-2n}) - \ldots$$

$$< 2^n - 2.$$

Thus, the only case where $E \leq 1$ is for $\delta = 0$ and $\lambda = 0$. We have proved the following theorem.

Theorem 1.

Given $n > 3$, the number of bits of the divisor and the dividend we must use to obtain n bits of the quotient at each iteration of Algorithm A, subject to the constraint that the sum of the number of bits looked at be minimum, is given by:

$$p = n + 3$$

and

$$m = n + 3,$$

where p is the number of bits of the dividend used and m is the number of bits of the divisor used.

If we wish to use values of p and m other than those in Theorem 1, the largest value of n acceptable can be calculated numerically. For an exact calculation, given values of p and m, we must examine equation 16.

We rewrite equation 16 as follows:

$$E = S - T,$$

where

$$S = \min\left(\left[\frac{a+1}{b}t\right]_{GIL}, 2^n - 1\right)$$

and

$$T = \left[\frac{a}{b+1}t\right]_{GI}.$$

Both S and T are step functions, monotone increasing in a and monotone decreasing in b. Let us examine S.

Assume that b is fixed and vary a. The values of S range from 0 to $2^n - 1$. Further, S will change value at some value of a, say x. For $a \leq x$, S will be one value and for $a > x$, S will be one greater. If y is the next value of a at which S changes values we can represent this arbitrary interval I, as follows:

$$I = (x,y].$$

On this interval, S is constant. T, however, may not be constant.

T is monotone increasing on this interval. Since E = S - T, E will assume the largest value, for a value of a approaching a from above. In fact, since a is an integer, this will be at a point $a = [x]_{LIG}$, (if such an integer exists in the interval I).

The values of x where S changes value are $x = \frac{kb}{t} - 1$, and k is a positive integer. This can be seen by evaluating S at a = x.

$$S = \left[\frac{a+1}{b}t\right]_{GIL} = \left[\frac{kb}{b}\right]_{GIL} = k - 1.$$

The values of a we need consider are $a = [x]_{LIG} = [x+1]_{GI} = \left[\frac{kb}{t}\right]_{GI}$. We need not consider values of $k > 2^n - 1$, since S is never greater than $2^n - 1$. Further, since $x \geq 0$, we need not consider values of k less than one.

The error E, of our quotient estimates, can now be evaluated using equation 16. We must check all values of $b \in [2^{m-2}, 2^{m-1})$ and values of $a = \left[\frac{kb}{t}\right]_{GI}$, where $k \in [1, 2^n - 1]$. An algorithm to determine the largest value of n acceptable for given values of p and m follows.

ALGORITHM B. Given values of p and m, and assuming the dividend and the divisor are positive, find the largest value of n such that the error $E \leq 1$ and equation 16 holds.

B1. [Initialize] Set $n \leftarrow \max\left(0, \log_2\left(\frac{2^{p-2}(2^{m-2}+1)}{2^{p-2}+2^{m-2}+1}\right)\right)$ (this

value is a lower bound and thus a good starting point).

B2. [Try a larger n] Set $n \leftarrow n + 1$. (Try this value of n. If it fails give $E < 1$ for all cases when the last value of n is the solution).

B3. [Divide into cases] If $2^{n+m} < 2^p$ go to B7.

B4. [Check] Set $t \leftarrow 2^{m+n-p}$. For all integral values of b in the range $b \in [2^{m-2}, 2^{m-1})$ and for all integral values of k in range $k \in [1, 2^n - 1]$ perform B5. When finished go to B2.

B5. [Check E] Set $a \leftarrow \left[\frac{kb}{t}\right]_{GI}$. If $\min\left(\left[\frac{(a+1)t}{b}\right]_{GIL}, 2^n - 1\right) - \left[\frac{at}{b+1}\right]_{GI} > 1$ go to B6.

B6. [Finished] $n \leftarrow n - 1$. The result is n. Stop.

B7. [Check] Set $u \leftarrow 2^{p-m-n}$. For all integral values of b in the range $b \in [2^{m-2}, 2^{m-1})$ and for all integral values of k in the range $k \in [1, 2^n - 1]$ perform B8. When finished go to B2.

B8. [Calculate E] Set $a \leftarrow [kbu]_{GI}$. If $\min\left(\left[\frac{a+1}{bu}\right]_{GIL}, 2^n - 1\right) - \left[\frac{a}{(b+1)u}\right]_{GI} > 1$, go to B6.

This algorithm is divided into two parts because equation 16 takes on values of 2^j where j can be either positive or negative. If j is negative $u = 1/t$ is substituted in equation 16 so that the operations involve only integer arithmetic. Clearly this algorithm

Table 1

Maximum number bits developed for p and m.

p \ m	3	4	5	6	7	8	9	10	11	12
10	2 1.56	2 2.29	3 3.12	4 3.99	5 4.87	5 5.70	6 6.42	7 7.00	7 7.42	7 7.68
9	2 1.55	2 2.27	3 3.07	4 3.91	4 4.71	5 5.43	6 6.00	6 6.42	6 6.67	6 6.83
8	2 1.51	2 2.21	3 2.98	4 3.75	4 4.44	5 5.01	5 5.42	5 5.68	5 5.83	5 5.91
7	2 1.46	2 2.11	3 2.81	3 3.47	4 4.02	4 4.42	4 4.68	4 4.68	4 4.91	5 4.96
6	2 1.34	2 1.93	3 2.53	3 3.04	3 3.43	3 3.68	3 3.83	4 3.91	4 3.96	4 3.98
5	2 1.13	2 1.62	2 2.08	2 2.44	3 2.69	3 2.83	3 2.91	3 2.96	3 2.98	3 2.99
4	2 0.78	2 1.15	2 1.47	2 1.70	2 1.83	2 1.91	2 1.96	2 1.98	2 1.99	2 1.99
3	1 +0.26	1 +0.51	1 +0.71	1 +0.84	1 +0.91	1 +0.95	1 +0.97	1 +0.99	1 +0.99	1 +0.997
2	1 -0.42	1 -0.26	1 -0.15	1 -0.08	1 -0.04	1 -0.02	1 -0.01	1 -0.01	1 -0.00	1 -0.00
1	1 -0.22	1 -1.14	1 -1.08	1 -1.04	1 -1.02	1 -1.01	1 -1.00	1 -1.00	1 -1.00	1 -1.00

Explanation of entries in table.

1. Top entry is maximum allowable value of n.
2. Bottom entry is value of n from inequality 18.

will stop because for each value of n a finite number of checks are made.

The results of this algorithm for small values of p and m are listed in Table 1. For comparison purpose the lower bound for n calculating using the left-hand expression in inequality 18 is also given.

This concludes the analysis of the division algorithm for cases when $A \geq 0$. We now proceed to the case $A < 0$. The general procedure here is similar to the case $A \geq 0$ but there are a few differences.

From equation 14 we know that C is in the closed interval

$$C \in \left[\max\left(0,\ 2^n + \left[\frac{a}{b}t\right]_{GI}\right),\ 2^n + \left[\frac{a+1}{b+1}t\right]_{GIL}\right].$$

Our estimate c of C is the largest value C might be in this interval, viz.,

$$c = 2^n + \left[\frac{a+1}{b+1}t\right]_{GIL}.$$

As in the positive case, the worst error is obtained when C is the minimum value in the interval, which will be called C_{min}. The error in this case, is

$$E_m = c - C_{min}$$

$$= 2^n + \left[\frac{a+1}{b+1}t\right]_{GIL} - \max\left(0,\ 2^n + \left[\frac{a}{b}t\right]_{GI}\right). \qquad (25)$$

We approximate E_m by $\tilde{E}_m$ as follows:

$$E_m = 2^n + \left[\frac{a+1}{b+1}t\right]_{GIL} - \max\left(0,\ 2^n + \left[\frac{a}{b}t\right]_{GI}\right)$$

$$= 2^n + \left[\frac{a+1}{b+1}t\right]_{GIL} + \min\left(0,\ -2^n - \left[\frac{a}{b}t\right]_{GI}\right)$$

$$\leq 2^n + \frac{a+1}{b+1}t + \min\left(0,\ -2^n - \left[\frac{a}{b}t\right]_{GI}\right)$$

$$\leq 2^n + \frac{a+1}{b+1}t - 2^n - \left[\frac{a}{b}t\right]_{GI}.$$

Since $[x]_{GI} > x - 1$, we have

$$E_m < \frac{a+1}{b+1}t - \frac{a}{b}t + 1$$

or

$$E_m < \frac{b - a}{b(b + 1)}t + 1.$$

If we let $\tilde{E}_m = \frac{b - a}{b(b + 1)}t$ (26)

then we have $E_m < \tilde{E}_m + 1$. Applying the restriction $\tilde{E}_m \leq 1$, then we have

$$E_m < \tilde{E}_m + 1 \leq 2$$

which is equivalent to $E_m \leq 1$ since E_m is an integer.

The maximum value of $\tilde{E}_m$ can be found by considering equation 26. Since this function is decreasing in both a and b the maximum value of the function will be on the right boundary, namely on the line $\frac{a}{b}t = -2^n$. In fact, the maximum will be at $a = -2^{p-2}$ and $b = 2^{m-2}$.

Evaluating $\tilde{E}_m$ we find

$$\tilde{E}_m = \frac{2^{m+n-p} + 2^n}{2^{m-2} + 1}$$

applying the constraint $\tilde{E}_m \leq 1$ and solving for n we have

$$1 \geq \frac{2^{m+n-p} + 2^n}{2^{m-2} + 1}$$

$$2^n \leq \frac{2^{p-2}(2^{m-2} + 1)}{2^{m-2} + 2^{p-2}}$$

$$n \leq \log_2 \left[\frac{2^{p-2}(2^{m-2} + 1)}{2^{m-2} + 2^{p-2}}\right] . \quad (27)$$

Letting n be the largest possible value, we have

$$n = \left(\log_2 \left[\frac{2^{p-2}(2^{m-2} + 1)}{2^{m-2} + 2^{p-2}}\right]\right)_{GI} . \quad (28)$$

Comparing this value of n with the value calculated by equation 19 which is the value of n calculated in the case $A \geq 0$, we see that the value of n given by equation 19 is less than or equal to the value calculated by equation 28. Since we want a lower bound for n, equation 19 should be used. The proof of Theorem 1 is correct, even in the negative case, since equation 28 holds true for $p = n + 3$ and $m = n + 3$.

For an exact calculation of n when $a < 0$ we must use equation 25.

$$E_m = 2^n + \left[\frac{a+1}{b+1}t\right]_{GIL} - \max\left(0,\ 2^n + \left[\frac{a}{b}t\right]_{GI}\right) .$$

We can rewrite equation 25 as follows:

$$E_m = \left[\frac{a+1}{b+1}t\right]_{GIL} - \max\left(-2^n, \left[\frac{a}{b}t\right]_{GI}\right) .$$

Further, define S and T as follows:

$$S = \left[\frac{a+1}{b+1}t\right]_{GIL}$$

and

$$T = \max\left(-2^n, \left[\frac{a}{b}t\right]_{GI}\right) ,$$

then

$$E_m = S - T.$$

Both S and T are step functions, monotone increasing in a and monotone decreasing in b. Let us exam S.

Assume b is fixed and vary a. The values of S range from $-2^n + 1$ to zero. Further S will change value at some value of a, say x. For $a \leq x$, S will be one value and for $a > x$, S will be one greater. If y is the next value of a at which S changes values we can represent this interval I, as follows:

$$I = (x, y].$$

On this interval S is constant, however T may vary. T is monotone increasing on this interval. Since $E_m = S-T$, E_m will assume the largest value, for a value of a approaching x from above. In fact, since a is an integer this will be at a point $a = [x]_{LIG} = [x+1]_{GI}$. The values of x where S changes value are $x = \frac{k(b+1)}{t} - 1$, where

$k = -1, -2, -2, \ldots, -2^n + 1$. This can be seen by evaluating S at $a = x$.

$$S = \left[\frac{a+1}{b+1}t\right]_{GIL} = [k]_{GIL} = k - 1.$$

We need not consider values of $k < -2^n + 1$, since S is never less than -2^n.

The values of a we need to consider are at $a = [x+1]_{GI} = \left[\frac{k(b+1)}{t} - 1 + 1\right]_{GI} = \left[\frac{k(b+1)}{t}\right]_{GI}$. Since $a < 0$, we see that k is always less than zero. The error E_m, of our quotient estimates, can now be evaluated using equation 25. We must check all values of $b \in [2^{m-2}, 2^{m-1})$ and values of $a = \left[\frac{k(b+1)}{t}\right]_{GI}$, where $k \in [-2^n+1, -1]$.

An algorithm to determine the largest value of n acceptable for given values of p and m follows:

ALGORITHM C. Given values of p and m, and assuming the divisor is positive and the dividend is negative (as a result of the last iteration of the division algorithm), find the largest value of n such that $E_m \leq 1$ and equation 25 holds.

C1. [Initialize] Set $n \leftarrow \max\left\{0, \log_2\left[\frac{2^{p-2}(2^{m-2}+1)}{2^{p-2}+2^{m-2}+1}\right]\right\}$.

C2. [Try a larger n] Set $n \leftarrow n + 1$.

C3. [Divide into cases] If $2^{n+m} < 2^p$, go to C7.

C4. [Check] Set $t \leftarrow 2^{m+n-p}$. For all integral values of b in the

range $b \in [2^{m-2}, 2^{m-1})$ and for all integral values of k in the range $k \in [1, 2^n-1)$ perform C5. When finished go to C2.

C5. [Calculate E_m] Set $a \leftarrow \left[-\frac{kb}{t}\right]_{GI}$. If $\left[\frac{a+1}{b+1}t\right]_{GIL}$ - $\max\left(2^n, \left[\frac{a}{b}t\right]_{GI}\right) > 1$, go to C6.

C6. [Finished] Set $n \leftarrow n - 1$. The result is n. Stop.

C7. [Check] Set $u \leftarrow 2^{p-m-n}$. For all integral values of n in the range $b \in [2^{m-2}, 2^{m-1})$ and for all integral values of k in the range $k \in [1, 2^n-1]$ perform C8. When finished go to C2.

C8. [Calculating E_m] Set $a \leftarrow [-kbu]_{GI}$. If $\left[\frac{a+1}{(b+1)u}\right]_{GIL}$ - $\max\left(-2^n, \left[\frac{a}{bu}\right]_{GI}\right) > 1$, go to C6.

The results of the application of algorithm C for small values of p and m are displayed in Table 2. The lower bound for n as calculated by the expression on the right side of equation 27 is also displayed for comparison.

Our final choice for n must be a value which is less than or equal to the values calculated by both algorithms B and C. Comparison of Tables 1 and 2 indicated the values specified by Table 1 satisfy this. This observation is only valid for the range of p and m calculated. In the general case both algorithm B and C must be applied to find the largest acceptable value of n for a given p and m.

Table 2

Values of n, given p and m fixed with negative dividend.

p \ m	3	4	5	6	7	8	9	10	11	12
10	2 1.57	2 2.30	3 3.13	4 4.00	5 4.87	5 5.70	6 6.43	7 7.01	7 7.42	7 7.68
9	2 1.56	2 2.28	3 3.08	4 3.92	4 4.72	5 5.44	6 6.01	6 6.42	6 6.68	6 6.83
8	2 1.54	2 2.23	3 3.00	4 3.77	4 4.46	5 5.02	5 5.43	5 5.68	5 5.83	5 5.91
7	2 1.50	2 2.15	3 2.85	3 3.50	4 4.04	4 4.44	4 4.69	4 4.84	4 4.92	5 4.96
6	2 1.42	2 2.00	3 2.58	3 3.09	3 3.46	3 3.70	3 3.84	4 3.92	4 3.96	4 3.98
5	2 1.26	2 1.74	2 2.17	3 2.50	3 2.72	3 2.85	3 2.92	3 2.96	3 2.98	3 2.99
4	2 1.00	2 1.32	2 1.58	2 1.77	2 1.87	2 1.93	2 1.97	2 1.98	2 1.99	2 2.00
3	2 0.58	2 0.74	2 0.85	2 0.92	2 0.96	2 0.98	2 0.99	2 0.99	2 1.00	2 1.00
2	1 0	1 0	1 0	1 0	1 0	1 0	1 0	1 0	1 0	1 0
1	1 -0.74	1 -0.85	1 -0.92	1 -0.96	1 -0.98	1 -0.99	1 -0.00	1 -1.00	1 -1.00	1 -1.00

Explanation of entries in table.

1. Top entry is maximum allowable value of n.
2. Bottom entry is value of n from inequality 26.

Hardware Implementation

Before discussing a hardware implementation of the division scheme we must first consider the carry propagate adder in order to obtain the information on the time delay involved in adding two n bit numbers. To be general, considerations will be in terms of gate delay time, rather than specifying absolute times for some particular type of logic.

The basic logic equations involved in addition for the kth bit of two numbers are:

$$S_k = A_k \oplus B_k \oplus C_{k-1}$$

and

$$C_k = A_kB_k + (A_k + B_k)C_{k-1}$$

where A_k and B_k are the kth bits of the two numbers A and B, to be added, C_{k-1} is the carry out of the $k\text{-}1^{st}$ stage, $\oplus$ designates the exclusive or function, + the logical or operation and juxtaposition the logical and operation.

Connection of n stages of logic like this would result in a conventional n bit adder. If we assume the propagation delay of an exclusive or gate is two gate times, and that logical and and logical or require one gate delay we can calculate the time needed to add two n bit numbers. It takes two gate delays to calculate C_k from A_k, B_k and C_{k-1} and given C_{k-1} it takes two gate delays to

calculate S_k. Thus for two N bit numbers it takes 2N + 2 gate delays to calculate the sum.

Use of the carry propagate adder is a method to reduce the time required to add two numbers. While this method is a standard approach, I will derive it here, in order to obtain equations for the propagation delay as a function of the number of bits to be added. If we define two terms, the carry propagate, P_k, and the carry genrate G_k for each stage of the adder as follows:

$$P_k = A_k + B_k,$$

$$G_k = A_k B_k,$$

and define $G_0 = C_0$, then

$$C_k = G_k + P_k C_{k-1}$$

$$= G_k + P_k G_{k-1} + P_k P_{k-1} C_{k-2}.$$

By induction we have

$$C_{k+1} = \sum_{i=0}^{k} \prod_{j=i+1}^{k} P_j G_i \quad . \tag{29}$$

This allows calculation of C_{N+1} in three gate delays, one to generate the P's and G's and two to generate C_{N+1} from the P's and G's. The sum can be generated in two gate delays. The net time, then is five gate delays. The only problem is that for large values of N, the number of inputs required on a gate becomes excessive. While the use of equation 29 to generate carries is impractical, we can use this, with some modification as the basic building block to form the

carry propagate adder. The process will be to use equation 29 to generate the carries for a four bit adder and then to use the four bit adder as a basic building block to generate larger adders. The carry propagation between each four bit block is done by additional circuitry to speed the carry propagation. To do this we define some more terms. First we define the level one carry propagate term P^I_k.

$$P^I_k = \prod_{i=4k-3}^{4k} P_i$$

and the level one carry generate term G^I_k,

$$G^I_k = \sum_{i=4k-3}^{4k} \prod_{j=i+1}^{4k} P_j G_i \quad .$$

Using these terms we can derive an expression for C_{4m} as

$$C_{4m} = \sum_{i=0}^{4k} \prod_{j=i+1}^{4k} P_j G_i$$

$$= \sum_{i=0}^{4} \prod_{j=i+1}^{4} P^I_j G^I_i \quad . \tag{30}$$

Each P^I_k can be generated in two delays and the G^I_k is three gate delays. Each C_{4m} can be generated in two gate delays from the P^I's and the G^I's. Thus, C_{4m} is generated in five gate delays from any input to the adder. These carries (C_{4m}'s) represent the carry in to the four bit adder blocks. Any carry in the block can be generated in two gate delays from the carry in gate delays.

The same problems can appear here as did with the use of

equation 29. We must limit the size of the adder to avoid an excessive number of inputs to any one gate. We used equation 29 to generate carries for a four bit adder, likewise it is practical to limit the use of equation 30 to generating carries for four four bit blocks, or for a 16 bit adder.

To increase the size of the adder further and still limit the number of inputs to any one gate we add another level of lookahead. Similar to what was done in building the 16 bit adder out of the four four bit adders we define some more terms, first a level two carry propagate term P^{II}_k,

$$P^{II}_k = \prod_{i=3k+1}^{4k} P^I_i ,$$

and second, a level two carry generate term G^{II}_k,

$$G^{II}_k = \sum_{i=3k+1}^{4k} \prod_{j=i+1}^{4k} P^I_j G^I_i .$$

With these terms we generate C_{16m},

$$C_{16m} = \sum_{i=0}^{16m} \prod_{j=i+1}^{16m} P_j G_i$$

$$= \sum_{i=0}^{4m} \prod_{j=i+1}^{4m} P^I_j G^I_i$$

$$= \sum_{i=0}^{4} \prod_{j=i+1}^{4} P^{II}_j G^{II}_i .$$

We can see that C_{16m} is generated in two gate delays from the P^{II}'s

and G^{II}'s, which are generated in two gate delays from the P^{I}'s and G^{I}'s, which were generated in three gate delays from any input to the adder. Thus, C_{16m} is generated from the inputs to the adder in seven gate delays. In the same fashion, addition of the kth level of lookahead,enables C_{m4^k} to be generated in two gate delays longer than the generations of $C_{4m^{k-1}}$ or in the 2k+3 gate delays. If we consider the time from generating C_{16m} to any output (sum) the worst case would be for a bit in the position 16m - 1. In this case, the carry in from the level two lookahead must go through both a level zero and a level one lookahead carry generator, and then through an exclusive or gate to generate the sum, in other words six gate delays.

In general, with the addition of the kth level of lookahead, the longest delay in obtaining the sum will be for a carry in the position $m4^k - 1$, in which case we must propagate a carry through levels k-1, k-2, ..., 0 of the carry lookahead logic as well as through an exclusive or gate to get the sum. This amounts to 2k + 3 gate delays.

In summary with the addition of the kth level of the lookahead the longest delay is 2k + 2, the time to propagate the carry; plus 2k + 3, the time to generate a carry, for a total of 4k + 5 gate delays.

With level one lookahead carry logic we could have an adder of up to 16 bits. Without increasing the number of inputs to any gate,

using the level two lookahead carry logic we can generate an adder of four blocks of 16 bits or 64 bits. If we use level k lookahead logic, we can have up to 4^{k+1} bits in our adder. Rephrasing this, if we wish to have an N bit adder, it will take

$\left[\log_4 \left(\frac{N}{4}\right)\right]_{LIG} = [\log_4(N) - 1]_{LIG} = [\log_4(N)]_{GI}$ levels of lookahead

to form a complete carry propagate adder. The delay for an N bit adder will then be $4[\log_4 (N)]_{GI} + 5$ gate delays.

It should be noted that the equations for the carry propagate term P^k and the carry generate term G^k and the equation for the carries C_{m4^k} are funtionally equivalent to those of the $k\text{-}1^{st}$ level, thus the adder is made out of a number of functionally equivalent blocks.

These calculations have been made, using the assumption that both the logical and and the logical or require one gate delay. For some types of logic currently available this is not true. For instance ECL (emitter coupled logic), enables the logical or to be calculated without any additional logic used. Thus the calculations are dependent on the type of logic used. The use of ECL results in a smaller number of gate delays to complete an addition. It is conceivable, in fact reasonable to expect advances in circuitry to enable execution of portions of the addition even faster.

We now consider the carry save adder (CSA). The carry save

adder is an adder which does not try to propagate the carry. Instead at the output of each stage we have both a sum term and a carry. Also we have three inputs, which can be considered to be two inputs and a carry in, or just three inputs. The rationale for a carry save adder is to permit adding more than two numbers together. As a simple example, if we wanted to add three binary numbers together, at each stage we would feed to a CSA the appropriate bit of each number. The result, the sum and a carry for each bit position could then be fed to a CPA (carry propagate adder) to add the sum and the carries together. By use of as many levels of CSA's as necessary we can add an arbitrary number of numbers together, and only have to propagate the carry once in the final CPA. Schematically, then to add six numbers we could proceed as shown in Figure I.

We next consider the logic for each bit of a carry save adder with three inputs. If we consider the three inputs to be I_k, J_k and L_k and the outputs to be S_k and C_k then we have

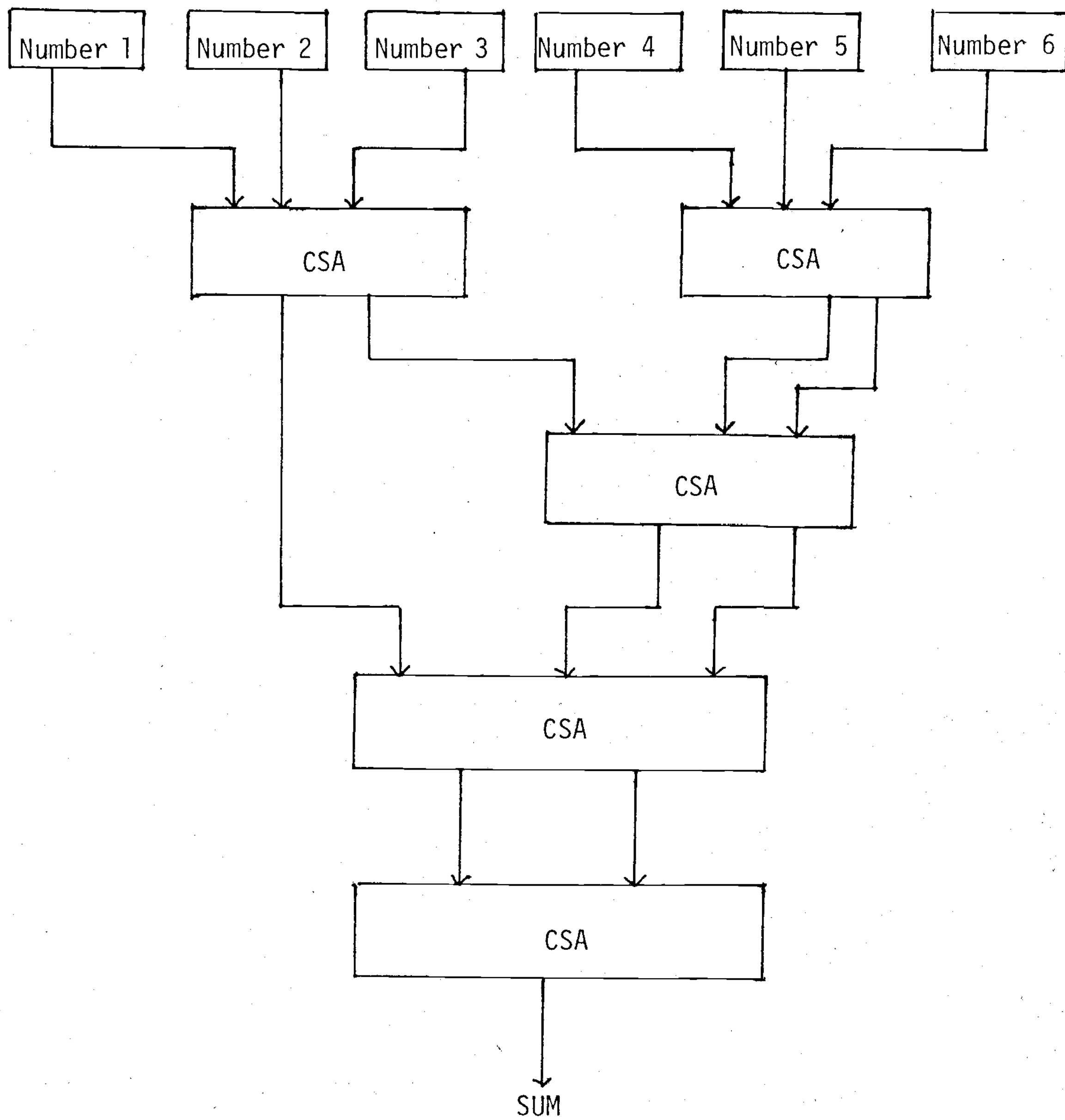
$$S_k = I_k \oplus J_k \oplus L_k$$

$$C_k = I_k L_k + J_k L_k + I_k J_k$$

so both S_k and C_k can be generated in two gate delays. With this information on the timing of the CSA and the CPA we can compare the results of the procedure for division described here with the other procedures.

Figure 1

Adding six numbers with a carry save adder tree.



An important consideration in a hardware implementation of the procedure is to choose the number of bits of the quotient n, obtained at each iteration to be as large as possible, without being so large as to be cost-prohibitive. By today's standards, as indicated by implementations of parallel multiplication by very similar logic, one would assume that n = 6 would be that magic number. Obviously this number is highly dependent on the state of the art, as well as being dependent on the cost vs. speed trade offs in any particular application.

Once the cost vs. speed trade offs are decided upon, Tables I and 2, or an extention of them can be used to choose a value of n in designing a dividing circuit. Or more reasonably, they can be used to choose a few values of n to investigate further. Tables I and 2 give values for p and m which will be necessary to obtain n bits of the dividend.

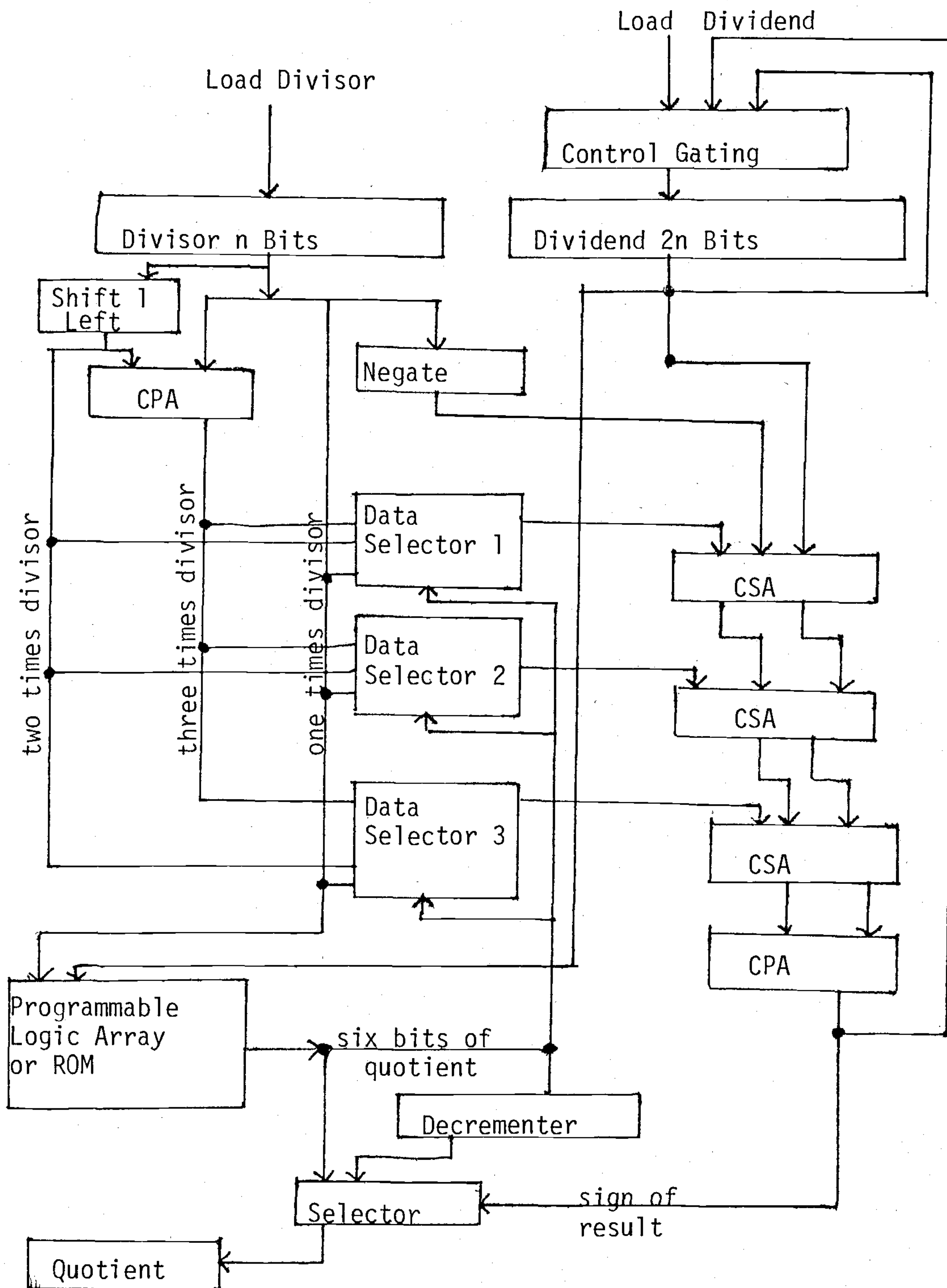
The n bits of the quotient would then be derived by table lookup or by logical derivation. If a table lookup is used 2^{m+p-2} entries are needed in the table, or $n2^{m+p-2}$ bits of information. Two methods currently available to implement this are the programable logic array (PLA) or a read only memory (ROM).

As an example of implementation, consider the case when n = 6. Once six bits of the quotient have been obtained the actual division procedure is simple. Figure 2 illustrates the general purpose.

The divisor and the dividend are stored in registers, the

Figure 2

Six bit division example.



dividend fed into a carry propagate adder to form three times the divisor. Since two times the divisor is merely a shift we now have available 0, 1, 2, and 3 times the divisor. The six bits of the quotient are used to control three data selectors, two bits going to each selector. The output of each selector will be either 0, 1, 2 or 3 times the divisor depending on the value of the control bits. The outputs of these data selectors are added together in a multiple input adder. By shifting the outputs of the data selectors before inserting the outputs into the adder, multiplication by a power of two can be obtained. Here we apply data selector one directly, representing 0, 1, 2, and 3 times the divisor. Data selector two is shifted left two binary places, thus representing 0, 4, 8, and 12 times the divisor. Data selector three is shifted four binary places left, representing 0, 16, 32, and 48 times the divisor. The net result is to obtain any multiple of the divisor from zero to 63 from the data selectors.

The multiple input adder consists of three carry save adder stages and one carry propagate adder stage. The output of the CPA is gated back into the dividend register and shifted left six places.

The six quotient bits are fed into an adder which subtracts one from the quotient. Either the six bits of the quotient generated by the programmable logic array (or ROM) or that value minus one is then gated to the quotient register using the sign of the number output from the CPA to select the appropriate value.

Part of the control logic counts the number of iterations and stops the process at the appropriate time. If the number in the dividend register is positive when the operation is finished, it is the remainder. If it is negative, we can obtain the remainder by adding the divisor to the number in the dividend register.

In order to compare the speed of this method with other existing methods we must first calculate the execution time for this method. We will consider the six bit implementation shown in Figure 2. We can calculate the execution time for a division of a 2N - 1 bit quotient by a N bit divisor. Let $k = [\log_4 N]_{GI}$. Assuming we have the divisor and the dividend loaded into the appropriate registers we can itemize and add up the delay times for each step of the operation. In order to initialize the process we must perform one add which is necessary to form the multiplies of the divisor needed. Once this initialization has been performed the rest of the calculations are in a loop, which generates six bits of the quotient at each iteration. Thus we need only calculate the time needed for one iteration and multiply by $\frac{N}{6}$, the number of times we go through the loop.

The time delay for developing the next six bits of the quotient depends upon the method used. State of the art programmable logic arrays would take the equivalent of seven gate delays. Read only memories could take almost any amount of time from three gate delays to thousands of gate delays depending on the technology involved.

Since the application here requires speed, a logical choice would be one of the faster ROM's requiring three to seven gate delays to read the information. Direct logic derivation could be done in three levels of logic. In general, three gate delays is a minimum and seven is a reasonable upper bound for this time delay.

The delays in the loop are as follows: seven gate delays maximum for the table lookup of the quotient digit; three gate delays for the data selectors, which choose the correct multiples of the divisor to feed to the adders; six gate delays to go through the CSA'a and 4k + 5 gate delays in the CPA, the final stage of the adder. The total gate delays in the loop is then 4k + 21, resulting in a total execution time of

$$4k + 5 + \frac{N}{6}(4k + 21).$$

We see that here, even with the added delay of the carry save adder tree the major portion of the time involved in the division process is due to the delay of the carry propagate adder. In fact a reasonable approximation is to say that the division time is $\frac{N}{6}$ times the time needed to add two numbers in the carry propagate adder, or simply $\frac{N}{6}$ times the add time.

Implementation of the Division Procedure in Software

While the method described in this paper is primarily for hardware implementation it can also be used in software to do multi-precision division operations. In this discussion the following assumptions are made: The divisor is assumed to be in normalized form, and the numbers are stored in k bit words, the most significant bit (sign bit) of the first word is used as the sign of the number, and the sign bit of each additional word of the number is set to zero. The digits of the number are stored in the lower k - 1 bits of each number. Further, we assume the computer has a divide instruction which divides a one word divisor into a two word signed dividend giving as results a one word signed quotient and a one word signed remainder.

The approach here is to use the hardware divide operation to obtain the next k - 1 bits of the quotient. We want k - 1 bits at each iteration since this is the number of bits stored in each word. If we obtain more or fewer than k - 1 bits, we necessitate shift operations on the numbers. If we can obtain exactly k - 1 bits at each iteration, we can eliminate the need to do shifts of the multi-precision numbers during the process.

We can use equation 19 to obtain an estimate of how many bits we can obtain at each iteration. We use equations 11 and 15 to obtain estimates of the quotient. Let b be in the first word of the

divisor, then m, the number of bits in b including the sign and the first bit of the number, will be k. Likewise, if we form a from the first two words of the dividend, then p, the number of bits of a, will be 2k - 1, since the sign bit of the second word of the dividend contains no information. With this information about the estimates of the divisor and the dividend we can evaluate n, the number of bits of the quotient obtainble at each iteration using equation 19 with p = 2k - 1 and m = k as follows:

$$n = \left[\log_2 \left(\frac{2^{2k-3}\ (2^{k-2} + 1)}{2^{2k-3} + 2^{k-2} + 1}\right)\right]_{GI}$$

$$= \left[-\log_2 \left(\frac{1}{2^{2k-3}} + \frac{1}{2^{k-2} + 1}\right)\right]_{GI} .$$

For $k \geq 3$ this becomes

$$n = k - 2.$$

Since we would like to obtain k - 1 bits of the quotient at each iteration this result is not desirable. In order to obtain k - 1 bits of the quotient we must have m = k + 1, thus b, the divisor in the calculation of the next quotient digit, must be k + 1 bits long.

Since we assume a hardware divide instruction, which used a k bit number of the divisor, it appears that we cannot obtain a k - 1 bit quotient in this manner. However, a minor adjustment will obtain the desired results.

Considering the problem of the dividing a k + 1 bit divisor B,

into a 2k - 1 bit dividend A. We consider the binary point of the divisor to be located after the kth bit (from the left). Thus, B can be broken up into an integer, B' and a fraction $\delta \in \{0, 1/2\}$. Using the hardware divide instruction we can divide B' into A obtaining a quotient Q', and a remainder R'.

$$\frac{A}{B'} = Q' + \frac{R'}{B'} \text{ or } A = Q'B' + R'.$$

First consider the case $A \geq 0$, and $B > 0$. Form $Q' = \left[\frac{A}{B'}\right]_{GI}$ and $R' = A - Q'B'$, $0 \leq R' < B'$ using the hardware divide instruction. We calculate R

$$R = A-Q'B = A-Q'(B'+\delta) = A-Q'B'-Q'\delta = R'-Q'\delta.$$

If $\delta = 0$, then $R = R'$. Since $B = B' > R' = R$, we have $B > R \geq 0$. Thus Q' is the correct quotient.

Assume $\delta = 1/2$. We have $R = R' - 1/2\ Q'$. If $R \geq 0$, then Q' is the correct quotient. However, if $R < 0$, then Q' is not correct. We must try Q' - 1 as the quotient. We calculate

$$\begin{aligned} R &= A - (Q' - 1)B = A - (Q' - 1)(B' + 1/2) \\ &= A - Q'B' - 1/2\ Q' + B' + 1/2 \\ &= R' - 1/2\ Q' + B' + 1/2 \\ &\geq R'. \end{aligned}$$

The last two inequalities are valid because

$$2^{k-1} \leq B' \leq 2^k - 1$$

and

$$0 \leq Q' < 2^{k-1}.$$

This gives the results $2B' > Q'$. Since $R \geq R'$, we know that $R > 0$ and that $Q = Q' - 1$ is the quotient.

In summary, to perform the division $\frac{A}{B}$, $B = B' + \delta$, where B' is an integer and δ is a fraction, $\delta \in \{0, 1/2\}$ as described above. We estimate the quotient q of $Q' = \left[\frac{A}{B'}\right]_{GI}$ and obtain the remainder $R' = A - Q'R'$ using the hardware divide instruction of the computer.

If $\delta = 0$ we are done, Q' is the quotient and R' is the remainder. Otherwise $\delta = 1/2$ and we calculate

$$R = R' - Q'/2.$$

Note, this is only a shift and an addition. If R is positive the Q' is the quotient and R is the remainder. If R is negative, the quotient is $Q' - 1$ and the remainder is $R - B' + \delta$. We have found

$$Q = \left[\frac{A}{B}\right]_{GI}.$$

These operations are simple adjustments to make to the results of the hardware division instruction to obtain the added precision dividend.

The case, $A < 0$, is similar and can be made, but it is equivalent to changing the sign of A and using the same procedure as used for the positive case.

With this extended precision divide, the multi-precision divide can be easily implemented on any binary computer.

The quotient estimate of the next k-1 bits of the quotient are

obtained with one divide and in the worst case a shift, two subtractions and an addition. Each iteration of the division can be completed with one multi-precision multiply and one multi-precision add of this product to the dividend.

Summary

I have defined an algorithm for doing division on a digital computer. The method is such that it can use the circuitry that is used to perform multiplication with very little modification. The speed of the division operation is insignificantly longer than that of multiplication. The timing was derived for a division operation which developed six bits of the quotient at each iteration, the results were

$$4k + 5 + \frac{N}{6}(4k + 21),$$

where N was the word size and $k = [\log_4(N)]_{GI}$. The units of time were logic gate delays. The time to add two numbers of this length (N) using a carry save adder tree was

$$4k + 5.$$

For large values of N these results can be approximated by saying the division times is approximately $\frac{N}{6}$ times the time required to add two numbers of equivalent length.

The calculations were made only for a process developing six bits of the quotient at a time. If n bits were developed at each iteration the results would have been

$$4k + 5 + \frac{N}{n}(4k + 21 + \delta)$$

where δ is zero for n = 6, and will increase slightly as n increases. The addition of δ is necessary because of a longer propagation delay

in the carry save adder tree as the number of inputs increase. Thus, for large N, the division time is approximately $\frac{N}{n}$ times the time to add two numbers of equivalent length.

Mac Sorley [21] quotes execution times in his paper for division algorithm which uses the shift and add techniques. The times he quotes are the number of additions required, based on the premise that the addition is the significant part of the division process, and the other operations tend to be almost insignificant compared to it. The best results were to obtain, on the average 3.8 bits of the quotient per addition. The method described here performs the division in approximately N/n additions, obtaining roughly n bits of the quotient per addition.

Methods which use multiplication and Newton's iterative approximation of the reciprocal require one full precision multiplication of the reciprocal of the divisor by the dividend. These methods require more than the time required to do two multiplications compared to the method proposed here which requires only slightly more than the time to do one multiplication.

The Harvard iterative scheme is faster than the scheme using Newton's approximation. It allows two operations to be performed in parallel. Times quoted by Anderson [2] indicate division times equivalent to two full precision multiplications. This is about twice as long as the method proposed herein.

The key factor is that the method proposed here is inherently

faster than those using multiply as the iterative operations. It is also faster than those methods obtaining one or two bits at a time. Add to this the fact that the hardware to perform the division operation (Figure 2) is essentially the same as the hardware for multiplication, the difference being mostly the control logic and you have a reasonably fast and cost effective method to perform multiplication and division on a digital computer.

Finally,it was proven that in a non-restoring iterative division algorithm it is only necessary to consider $n + 3$ bits ($n > 3$) of the dividend and divisor to obtain a quotient estimator.

Bibliography

1. Ahmad, M., "Iterative Schemes for High Speed Division", Computer Journal, Vol. 15, No. 4, pp. 333-336, November 1972.

2. Anderson, S.F., Earle, J.G., Goldschmidt, R.E., and Powers, D.M., "The IBM System/360 Model 91 Floating-Point Execution Unit", IBM Journal of Research and Development, Vol. 2, pp. 34-53, January 1967.

3. Atkins, D.E., "Higher Radix Division Using Estimates of the Divisor and Partial Remainders", IEEE Trans. Comput., Vol. C-17, pp. 925-934, October 1968.

4. Atkins, D.E., "A Study of Methods for Selection of Quotient Digits During Digital Division", University of Illinois, Urbana, Illinoise, p. 128, Thesis 1970.

5. Atkins, D.E., "The Analysis and Design of a Class of Quotient Digit Selectors", 5th IEEE Int. Comp. Soc. Conf. on Hardware, Software and Firmware and Tradeoffs, pp. 201-202. September 1971.

6. Bennett, W.S., "Quotient Generation With Conventional Binary Multiplication", Proc. IEEE, Vol. 61, No. 5, pp. 664-665, May 1973.

7. Cappa, M., and Hamacher, V.C., "An Augmented Iterative Array for High Speed Binary Division", IEEE Trans. Comput., Vol. C-22, No. 2, pp. 172-175, February 1973.

8. Chu, Y., "Digital Computer Design Fundamentals", McGraw Hill, pp. 35-42, 1962.

9. Dean, K.J., "Cellular Arrays for Binary Division", IEE Proc., Vol. 117, pp. 917-920, May 1970.

10. Dean, K.J., "A Precision Code Convertor for Reciprocals of Binary Numbers", The Computer Bulletin, Vol. 12, No. 2, pp. 55-58, 1968.

11. Ferrari, D., "A Division Method Using a Parallel Multiplier", IEEE Trans. Comput., Vol. Ec-16, No. 2, pp. 224-226, February 1967.

12. Flores, Ivan, "The Logic of Computer Arithmetic", Prentice Hall, pp. 246-347, 1963.

13. Flynn, M.J., "On Division by Functional Iteration", IEEE Trans. Comput., Vol. C-19, No. 8, pp. 702-706, August 1970.

14. Freiman, C.V., "Statistical Analysis of Certain Binary Division Algorithms", Proc. IRE, Vol. 49, pp. 91-103, January 1961.

15. Gill, S.(1955), Internal Communications quoted from 1. Ahmad, M., "Iterative Schemes for High Speed Division", Computer Journal, Vol. 15, No. 4, pp. 333-336, November 1972.

16. Knuth, D.E., "The Art of Computer Programming", Vol. 2, Seminumerical Algorithms, Addison-Wesley, Reading Mass., pp. 229-245, 1969.

17. Krishnamurthy, E.V., "On a Divide-and-Correct Method for Variable Precision Division", CACM, Vol. 8, No. 3, pp. 179-181, March 1965.

18. Krishnamurthy, E.V., "On Optimal Iterative Schemes for High-Speed Division", IEEE Trans. Comput., Vol. C-19, No. 3, pp. 227-231, March 1970.

19. Krishnamurthy, E.V., "Economic Iterative and Range-Transformation Schemes for Division", IEEE Trans. Comput., Vol. C-20, No. 4, pp. 470-472, April 1971.

20. Krishnamurthy, E.V., and Nandi, S.K., "On the Normalization Requirement of Divisor in Divide-and-Correct Methods", CACM, Vol. 10, No. 12, pp. 809-813, December 1967.

21. Mac Sorley, O.L., "High-Speed Arithmetic in Binary Computers", Proc. IRE, VOL. 49, pp. 67-91, January 1961.

22. Mifsud, C.J., "A Multiple-Precision Division Algorithm", CACM, Vol. 13, No. 11, pp. 666-668, November 1970.

23. Mifsud, C.J., and Bohlen, M.J., "Addendum to A Multiple-Precision Division Algorithm", CACM, Vol. 16, No. 10, p. 628, October 1973.

24. Nandi, S.K., and Krishnamurthy, E.V., "A Simple Technique for Digital Division", CACM, Vol. 10, No. 5, pp. 299-301, May 1967.

25. Richards, R.K., "Arithmetic Operations in Digital Computers", London: Van Nostrand, pp. 279-281, 1955.

26. Riesel, Z., and Shahan, Z., "A Note on Division Algorithms Based on Multiplication", IEEE Trans. Comput., Vol. C-21, No. 5, pp. 513-514, March 1972.

27. Roberton, J.E., "The Correspondence Between Methods of Digital Division and Multiplier Recording Procedures", IEEE Trans. Comput., Vol. C-19, No. 8, pp. 692-701, August 1970.

28. Robertson, J.E., "A New Class of Digital Division Methods", IRE Trans., Vol. EC-7, pp. 218-222, 1958.

29. Stefanelli, R., "A Suggestion for a High-Speed Parallel Binary Divider", IEEE Trans. Comput., Vol. C-21, No. 1, pp. 42-55, January 1972.

30. Stein, M.C., "Divide-and-Correct Methods for Multiple Precision Division", CACM, Vol. 7, No. 8, pp. 472-474, August 1964.

31. Stein, M.C., and Pope, D.A., "Multiple Precision Arithmetic", CACM, Vol. 3, No. 12, p. 652, December 1960.

32. Von Neumann, J., Goldstine, H.H., and Burk, A.W., "Logic Designing of Electronic Computing Instruments", (1947) from "Collected Works, John Von Neumann", Vol. 5, pp. 35-79, MacMillan Company, New York, 1963.

33. Wallace, C.S., "A Suggestion for a Fast Multiplier", IEEE Trans. Comput., Vol. Ec-13, No. 1, pp. 14-17, February 1964.

APPENDICES

Appendix A

Index to Defined Symbols

The following is a list of the more important symbols used in this paper. It does not include symbols which are significant within only a small portion of the paper. Where possible a short description is given, not as a definition but rather as an indication of the symbols usage.

Symbol	Page	Defined Usage
A	11	Dividend
a	12	Estimate of Dividend
B	11	Divisor
b	12	Estimate of Divisor
C	10	Quotient Digit
C_{min}	22	
c	10	Estimate of C
E	22	Error
$\tilde{E}$	22	Approximate Error
E_m	36	Error
$\tilde{E}_m$	37	Approximate Erro
m	12	Size of b
n	9	Number of Bits Developed at Each Iteration
N	48	Word Length

Symbol	Page	Defined Usage
p	12	Size of a
Q	5	Quotient
t	13	2^{n+m-p}

Appendix B

Definition of Abbreviations

$[x]_{GI}$ = The greatest integer less than or equal to x.

$[x]_{GIL}$ = The greatest integer less than and not equal to x.

$[x]_{LIG}$ = The least integer greater than and not equal to x.