TWO SQUARE ROOT ALGORITHMS UTILIZING MULTIPLICATION AS THE ITERATIVE OPERATOR

University of Texas
NGR-44-012-144

APPROVED:

_____

_____

TWO SQUARE ROOT ALGORITHMS UTILIZING MULTIPLICATION AS THE ITERATIVE OPERATOR

by

JAMES RICHARD GOODMAN, B.S.

THESIS

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

THE UNIVERSITY OF TEXAS AT AUSTIN

August 1969

ACKNOWLEDGMENTS

The subject for this research was suggested by Professor C. V. Ramamoorthy. In addition, Professor Ramamoorthy provided invaluable assistance by his guidanc and encouragement. He made many suggestions which were of great help and for this the author wishes to express his sincere thanks.

The author wishes to thank Professor C. H. Roth, Jr., for his helpful suggestions and interest as a member of the committee.

The author also appreciates the guidance rendered by Professor David M. Young, Jr., in the convergence proofs.

J. R. G.

August, 1969

ABSTRACT

The hardware algorithms are presented which utilize high-speed multiplication and no division to perform the square root operation rapidly. One algorithm is intended for a large general-purpose computer and in addition provides a second-order division scheme. The second algorithm requires a special function generator which is presently utilized in certain existing computers. Each algorithm is considered for convergence rate, variance, accuracy and implementation. The effect and importance of the initial approximation is considered. A simulation is performed to compare each to a conventional algorithm. Although both algorithms are intended strictly for hardware implementation, either may find an application using microprogramming, and under certain conditions, one might be implemented in software.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

The increasing use of computers in scientifically oriented real-time applications such as process control, aerospace navigation and guidance, etc., has placed new demands on computational speed. Therefore, the frequently used software arithmetic operations such as division, square root, and possibly other functions using a polynomial evaluation have to be performed in the hardware.

In this thesis, we are concerned with generating and implementing the square root function, using new techniques which make use of fast multipliers found in large, fourth-generation scientific machines. [1], [3]

Most general purpose scientific machines such as the CDC 6600, SDS 930, and the Honeywell 8200 use programmed subroutines to evaluate square root. These subroutines, many of which employ an iterative division algorithm, are naturally very slow in comparison to multiplication. On the opposite extreme some computers such as the Philco Transac 1000 have a built-in square-root facility. Although an order of magnitude increase in speed is possible for such an addition, the cost of the additional hardware required can be prohibitive because

of the complex sequencing that is needed in the square root generation.

It is felt that, somewhere between these extremes lies a host of computational techniques which reduce execution time by making use of the inherent parallelism of the algorithms. Hopefully they could also decrease cost of additional equipment by maximum utilization of existing hardware.

The approach of LSI technology promises a reduction in logic cost in addition to speed and size improvements. It also makes more attractive specialized arithmetic function generators due to the resultant emphasis on functional partitioning. It is believed, therefore, that a technique as mentioned above, if implementable with LSI-type logic arrays, will be a practical approach to increasing computation speed for the square-root function.

CHAPTER 2

SQUARE ROOT TECHNIQUES

2.1 Direct Methods

So far as is known, the only hardware method used to date to generate square root is a binary equivalent of the paper-and-pencil method everyone learns in grade school. The paper-and-pencil method is as follows:

> Pair the digits off both directions from the decimal point. Starting with the left-most pair of digits (or single digit) find the largest number which, when squared, is less than the digit pair. Subtract the square from the digit pair and append the next digit pair. Call this the remainder. The first digit found is the first digit of the answer. The second digit is determined in the following way: Double the answer so far calculated (so far only one digit). Append to that number the largest digit for which the product of the digit and the appended number (i.e., 20 times the answer plus the digit) is less than the remainder. Subtract that product from the remainder and append the next decimal pair.
>
> The remaining digits are calculated just as the second one was--finding the largest digit which, when

added to 20 times the answer and multiplied by the digit is less than the remainder. The decimal point is after the same digit as the number of digit pairs to the left of the decimal point of the initial number. An example is given in Fig. 2-1 for the root of 62,138.

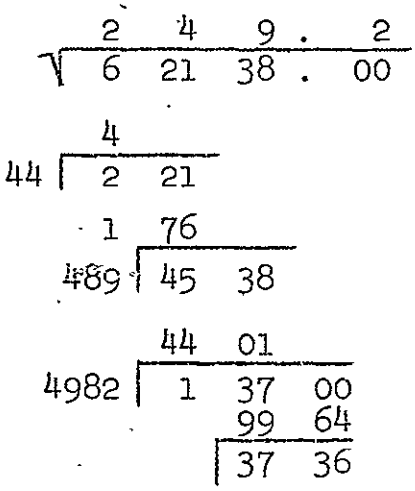


Fig. 2-1 Decimal Square Root.

The binary algorithm is quite analogous to the conventional decimal technique. It is as follows with example in Fig. 2-2.

Pair the number off in bits both ways from the binary point. If the number is in floating point the exponent must be an even number and the mantissa must be less than one and greater than or equal to 1/4. The first bit of the result must be one, since the first bit pair of the "normalized" number must be 11, 10, or 01. Subtract one from the first bit pair and append the next

bit pair to the difference. Double the answer so far calculated (in this case only the bit 1) and append the bit 1 if that number is less than the remainder. Append a 1 also to the answer. Subtract from the remainder and bring down the next bit pair. If the number calculated is less than the old remainder, append a 0 to the answer and bring down the next bit pair. The remaining bits are calculated just as the second bit was.

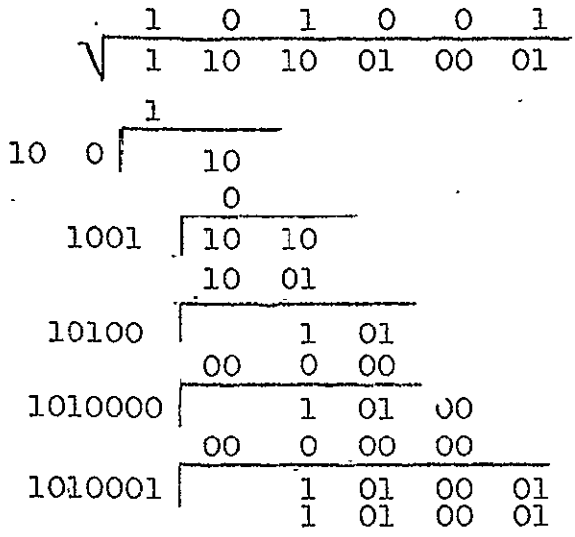


Fig. 2-2 Binary Square Root

In the machine it is difficult to look at the number and decide if it is larger or smaller than another number. Thus at each stage it is assumed that the next bit is a one and the number formed by a one bit appended to twice

the answer is subtracted from the remainder. If it is negative it is added back to the remainder and last answer bit is changed to zero. This is called the restoring square root algorithm. An example is in Fig. 2-3.

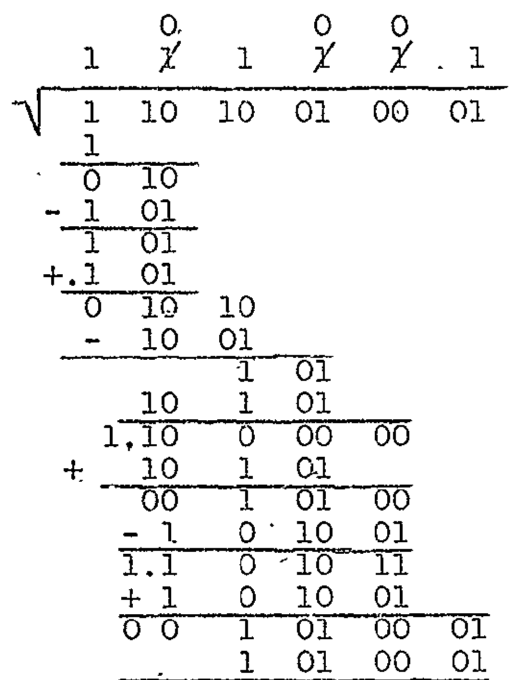


Fig. 2-3 Restoring Square Root

A variation of this algorithm which is faster but requires more hardware is the nonrestoring square root algorithm. In this method, again a one bit is assumed, but if the difference is negative the number is not added back in. Instead, on the following cycle the subtrahend is made by appending a pair of ones to the answer thus far, including the zero bit just determined. This number

then is added, rather than subtracted as would normally be done. This process saves a step in that it combines the addition with the next subtraction for each time the number would normally have to be restored. An example is given in Fig. 2-4.

```
          0       0   0
    1     X   1   X   X   1
  √ 1   10  10  01  00  01
    1
    --
      10
  - 1 01
  ---
  1.1 01  10
      10  11
  ---------
          1  01
    - 1   01  01
    -----------
    1.1   00  00  00
        + 10  10  11
    -------------------
      11  10  10  11  01
          1   01  00  11
    -------------------
          00  00  00  00
```

Fig. 2-4 Nonrestoring Square Root

The nonrestoring algorithm has been used more due to the fact that with little more hardware it is substantially faster. It is used, for example, in many airborne computers such as the Philco Transac-1000. For either algorithm additional hardware is required for sequencing. In addition a counter is necessary to determine completion, although presumably if a hardware division scheme is implemented using subtraction as the iterative operator, the same counter can be used for both

with slight modifications. The number of iterations required to complete the square root is fixed generally by the size of the operand.

2.2 Iterative Approximation Techniques

Due to the complexity of the sequencing of direct methods, most software routines have used certain iterative methods to generate square root. Two of the most common methods are briefly discussed below:

2.3 Approximation to the Taylor Square Root Polynomial

The number is classified in one of many ranges and then, using a number of constants, the first few terms of the polynomial are generated. The SDS 930, for example, uses eight ranges with four constants per range to generate a 20-bit square root. [6]

2.4 Newton-Raphson Iteration

After the generation of an initial approximation the Newton-Raphson iteration is used recursively. For a number $N = B^2$, initial approximation B_o, the Newton-Raphson iteration is of the form:

$$B_{k+1} = \frac{1}{2} \left(B_k + \frac{N}{B_k}\right)$$

It can be shown that the number of places of accuracy doubles with each iteration. The Control Data 6600

computer uses this method after generating an initial approximation with maximum relative error less than 2.6×10^{-3}. [3] The subroutine, in addition to requiring the storage of three constants requires over 200 clock periods to generate the initial approximation and go through the four Newton-Raphson iterations required.

The SDS 930, which uses the polynomial evaluation for single-precision calculations, utilizes the Newton-Raphson iteration for multiprecision work. Only one iteration is necessary, however, for double precision, since the polynomial evaluation has already generated a single precision initial approximation.

Each of these methods reveals inherent problems for hardware implementation. The polynomial approximation requires a large number of constants, the number of which increases faster than the word length. The Newton-Raphson technique requires a division on each iteration, which makes the iteration very slow. Division is a very slow operation. For example, in the CDC 6600 computer it takes 7.25 times as long as an addition and nearly three times as long as a multiplication. It is so slow, even compared to multiplication, that in some of the newer machines [1] it is being performed using multiplication as the iterative operator instead of the traditional subtraction.

Thus the software Newton-Raphson iteration becomes limited by the division rate.

It seems clear, therefore, that if an algorithm can be derived which eliminates division in the iteration, a considerable speed-up can be realized.

CHAPTER 3

ALGORITHM R

3.1 Motivation

If a simple method could be obtained to find the reciprocal of B_k, the Newton-Raphson iteration could be performed sans division. Clearly the Newton-Raphson iteration for the reciprocal could be nested in the square root iteration, i.e.,

given $\quad N = B^2,\ B_o \simeq B,\ R_{k,o} \simeq \frac{1}{B}$

$$\left[\begin{array}{ll} R_{k,j+1} = R_{k,j}(2-R_{k,j}B_k) & j=1,2 \ \ldots\ m \\ B_{k+1} = \frac{1}{2}(B_k+NR_{k,m}) & \end{array}\right]_{k=1,2,\ \ldots\ n} \qquad \begin{array}{l}(3\text{-}1)\\ (3\text{-}2)\end{array}$$

for sufficiently large m and n.

However, since two multiplications are required for each iteration of R, a large number of multiplications would be required per iteration of B.

Now suppose that only one iteration for R were made between calculations of B and that the previous value of R were used for the approximation. It intuitively seems wasteful to produce many iterations of R_k for early values of B_k when B_k is only a rough approximation. If

the two iterations converge at approximately the same rate, the accuracy of R_k would be of the same order as B_k after one iteration. This is the motive for Algorithm R which is:

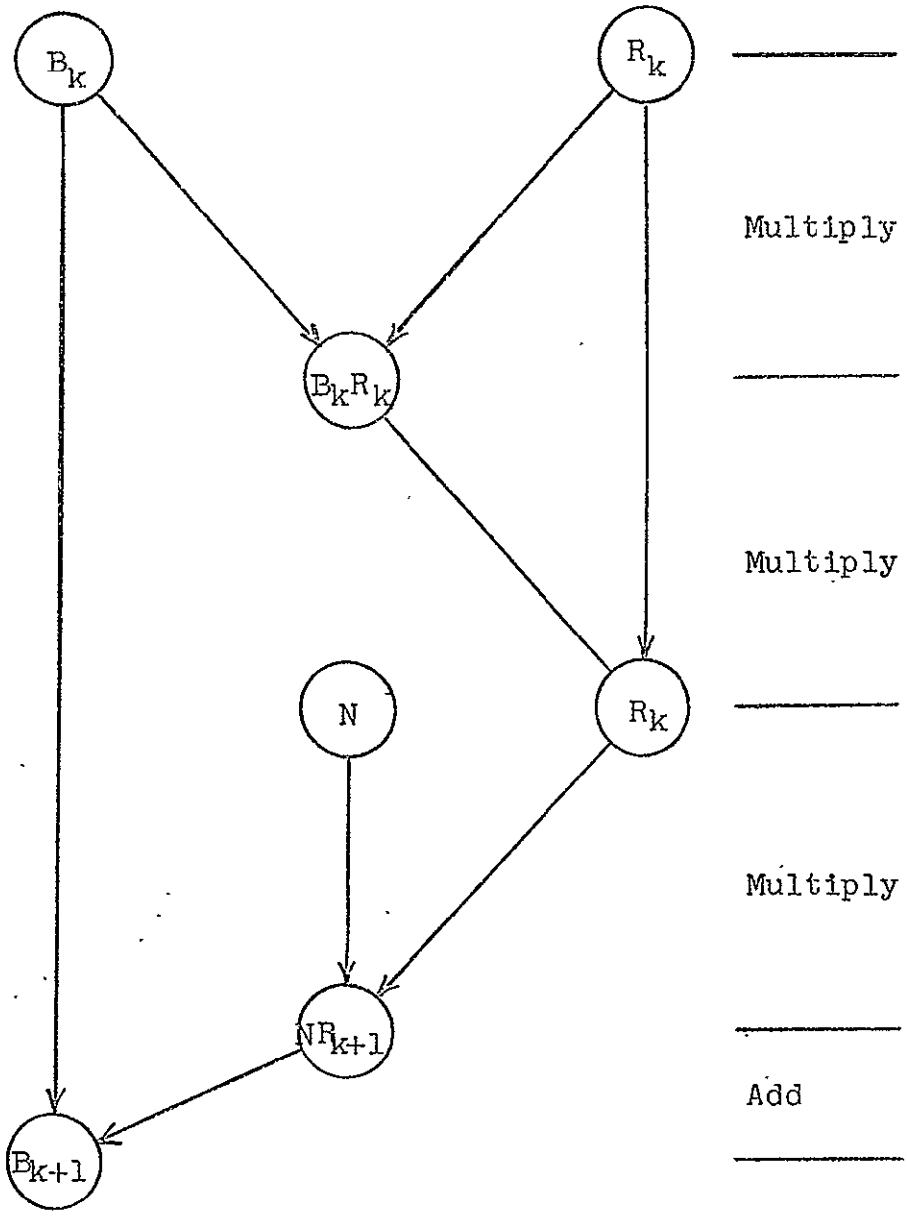
$$\text{Given} \quad N = B^2 = \frac{1}{R^2}\ , \ B_o \simeq B,\ R_o \simeq R$$

$$\text{The iteration} \quad R_{k+1} = R_k(2-B_kR_k) \tag{3-3}$$

$$B_{k+1} = \frac{1}{2}(B_k+NR_{k+1}) \tag{3-4}$$

converges rapidly to $B_m = B$, $R_m = R$. It will be shown later in the proof that this algorithm converges if the relative errors of B_o and R_o are each less than .208 .... Also in the proof the convergence rate will be calculated. It was assumed that some sort of a table look-up would be used to generate B_o and R_o to within specified tolerances.

A flow chart for algorithm R is presented in Fig. 3-1. Initially R_o and B_o are multiplied together. This product is then gated into the multiplier as a two's complement, forming the term $2-B_oR_o$, and multiplied by R_o, forming R_1. R_1 is then multiplied by N and added to B_o: If it is shifted left one place during the addition, it comes out as B_1. Thus an iteration requires 3 multiplication times plus an addition. With a little care, the addition can be fixed point, since the multiplicand and the multiplier were both normalized.



$$R_{k+1} = R_k\ (2 - B_kR_k)$$

$$B_{k+1} = \frac{1}{2}(B_k + NR_{k+1})$$

Fig. 3-1 Sequence Diagram for Algorithm R.

An example of algorithm R is given in Fig. 3-2. Along with the results of the calculations for B_k and R_k is given the relative errors $F_k = \frac{B_k - B}{B}$ and $E_k = \frac{R_k - R}{R}$. As shown in the Example $|E_5|$ is greater than E_4. That is, for this particular iteration R_k actually diverged slightly. This is not unusual. Both B_k and R_k occasionally diverge. However, just as in this example, whenever $|E_k|$ increases, $|F_k|$ decreases dramatically, and vice versa.

3.2 Division

A feature of the R-algorithm is that division can easily be performed using no additional hardware other than that required for algorithm R. Since R_{k+1} is the Newton-Raphson iteration for the reciprocal of B_k, if a number D is inserted in place of B_k and only the iteration for R_k is performed, R_k will converge at a second order rate to $R = 1/D$. If D is the divisor in a fraction $\frac{N}{D}$ the quotient may be determined by multiplying $\frac{N}{D} = N \times R$.

3.3 Proof of Convergence

The nondividing algorithm R converges to the square root of a number for an initial value sufficiently close. However, it does not converge for all initial values and

N=.45313

B(0)= .6875000000000	F(0)= 2.13E-02	R(0)= 1.500000000000	E(0)= 9.72E-03
B(1)= .6729772656250	F(1)=-2.56E-04	R(1)= 1.453125000000	E(1)=-2.18E-02
B(2)= .6729852071246	F(2)=-2.44E-04	R(2)= 1.485209870510	E(2)=-2.32E-04
B(3)= .6731492586061	F(3)=-8.35E-08	R(3)= 1.485916425943	E(3)= 2.44E-04
B(4)= .6731492948254	F(4)=-2.97E-08	R(4)= 1.485554545152	E(4)= 2.41E-08
B(5)= .6731493147883	F(5)=-1.06E-14	R(5)= 1.485554553332	E(5)= 2.97E-08
B(6)= .6731493147883	F(6)=-5.28E-15	R(6)= 1.485554509276	E(6)= 1.91E-14
B(7)= .6731493147883	F(7)=-5.28E-15	R(7)= 1.485554509276	E(7)= 4.78E-15
B(T)= .6731493147883		R(T)= 1.485554509276	

Fig. 3-2 An Example of Algorithm R

it does not converge uniformly. For example, if R_k ever happens to be exactly zero, then since it is a factor of R_{k+1}, R_{k+1} will be zero and R_k will converge to zero. If the product $R_k B_k$ is greater than two, R_{k+1} is negative and the sequence will diverge. However, for this algorithm, since the argument N is assumed to be normalized so that $1/4 \leq N < 1$, initial values may be assumed to be in the following ranges:

$$1/2 \leq B_o < 1 \qquad (3\text{-}5)$$

$$1 < R_o \leq 2 \qquad (3\text{-}6)$$

The proof will make the assumptions that the relative errors of R_o and B_o are less than $\frac{\sqrt{105} - 9}{6} = .208 \ldots$. It is assumed that any implementation using this method would have at least this accuracy in the initial approximation. The first bit of N, i.e., the information that $1 > N \geq 1/2$ or that $1/2 > N \geq 1/4$ is sufficient to generate an approximation of this accuracy. For example, if N is known to be between 1/2 and 1, the values $B_o = 2(\sqrt{2}-1) \simeq .828428$, $R_o = 2(2-\sqrt{2}) \simeq 1.171572$ each have relative errors of $\pm (3-2\sqrt{2}) \simeq \pm .17157$ for $N = 1$ and $N = 1/2$. The same is true for $1/4 < N < 1/2$ if $B_o = 2 - \sqrt{2} \simeq .585786$, $R_o = 4(\sqrt{2}-1) \simeq 1.656856$. If a five-bit approximation is used, the maximum relative error possible is $2^{-5} = \pm .03125$. Thus a five-bit initial approximation

of these values of B_o and R_o has a maximum relative error of less than .208 and will converge.

Similarly, if the two first bits of N are considered, values of R_o and B_o can be obtained with relative error of .072 or less. For this case, only a three-bit approximation to R_o and B_o is necessary to guarantee convergence.

The proof of convergence is not straightforward because neither of the sequences B_k nor R_k converges uniformly towards its limit. Thus, it was necessary to introduce a function ζ which is the sum of the magnitudes of the relative errors of B_k and R_k. The theorem and proof follow:

Theorem R: For a number $N = B^2 = \frac{1}{R^2}$, $\frac{1}{4} \leq N < 1$,

given a pair of numbers B_o and R_o

such that $\left|\frac{B_o-B}{B}\right|, \left|\frac{R_o-R}{R}\right| < \frac{\sqrt{105}-9}{6}$

and the iteration

$$R_{k+1} = R_k(2-B_kR_k) \tag{3-7}$$

$$B_{k+1} = \frac{1}{2}(B_k+NR_{k+1}), \tag{3-8}$$

$$\text{then } \lim_{k\to\infty} B_k = \lim_{k\to\infty} 1/R_k = \sqrt{N} \tag{3-9}$$

Proof: Define error functions as follows:

$$E_k = \frac{R_k-R}{R} \tag{3-10}$$

$$E_k = \frac{B_k-B}{B} \tag{3-11}$$

Substituting equation (1) into (2) gives

$$R_{k+1} = R_k(2-B_kR_k) = 2R_k - B_kR_k^2 \qquad (3\text{-}13)$$

$$B_{k+1} = \frac{1}{2}[B_k+N(R_k)(2-B_kR_k)] \qquad (3\text{-}14)$$

$$= NR_k + \frac{1}{2}B_k - \frac{1}{2}B_kR_k^2N \qquad (3\text{-}15)$$

$$B_{k+1} = B^2R_k + \frac{1}{2}B_k - \frac{1}{2}B_kR_k^2B \qquad (3\text{-}16)$$

$$E_{k+1} = \frac{R_{k+1}-R}{R} = \frac{(2R_k-B_kR_k^2)-R}{R} \qquad (3\text{-}17)$$

Substituting $R_k = (E_k+1)R = \frac{E_k+1}{B}$

and $B_k = (F_k+1)B = \frac{F_k+1}{R}$ gives

$$E_{k+1} = \frac{2E_kR+2R-(E_k^2F_k+2E_kF_k+F_k+E_k^2+2E_k+1)R-R}{R} \qquad (3\text{-}18)$$

which simplifies to

$$E_{k+1} = -F_k-E_k^2-2E_kF_k-E_k^2F_k \; . \qquad (3\text{-}19)$$

Likewise for F_{k+1}

$$F_{k+1} = \frac{B_{k+1}-B}{B} = \frac{(B^2R_k+\frac{1}{2}B_k-\frac{1}{2}B_kR_k^2B)-B}{B} =$$

$$\frac{\frac{B^2(E_k+1)}{B} + \frac{1}{2}B(1+F_k) - \frac{1}{2}B(E_k^2F_k+2E_kF_k+F_k+E_k^2+2E_k+1) - B}{B} \qquad (3\text{-}20)$$

which simplifies to

$$F_{k+1} = -\frac{1}{2} E_k^2 - E_k F_k - \frac{1}{2} E_k^2 F_k \qquad (3\text{-}21)$$

Define the composite error sum function

$$\delta_k = |E_k| + |F_k| \qquad (3\text{-}22)$$

It is clear that

$$\lim_{k \to \infty} \delta_k = 0 \qquad (3\text{-}23)$$

implies that $\lim_{k \to \infty} |E_k| = \lim_{k \to \infty} |F_k| = 0$ (3-24)

which will prove the theorem.

Using equations (3-19) and (3-21), the triangle inequality guarantees that

$$\delta_{k+1} = |F_{k+1}| + |E_{k+1}| \leq \frac{1}{2} |E_k^2| + |E_k F_k| +$$

$$\frac{1}{2} |E_k^2 F_k| + |F_k| + |E_k^2| + 2 |E_k F_k| + |E_k^2 F_k| \qquad (3\text{-}25)$$

$$\leq |F_k| + \frac{3}{2} |E_k^2| + 3 |E_k F_k| + \frac{3}{2} |E_k^2 F_k| \qquad (3\text{-}26)$$

$$\delta_{k+1} \leq |F_k| + (\frac{3}{2} |E_k| + 3 |F_k| + \frac{3}{2} |E_k F_k|) \cdot |E_k| \qquad (3\text{-}27)$$

Assuming that $|E_k|, |F_k| \leq \frac{\sqrt{105}-9}{6} = .208$

$$\delta_{k+1} \leq |F_k| + \left[\frac{3}{2}\frac{\sqrt{105}-9}{6} + 3\frac{\sqrt{105}-9}{6} + \frac{3}{2}\left(\frac{\sqrt{105}-9}{6}\right)^2\right] E_k \tag{3-28}$$

which simplifies to

$$\delta_{k+1} \leq |F_k| + 1\ |E_k| < \delta_k \tag{3-29}$$

If $\delta_k = 0$, $\delta_{k+1} = 0$. Otherwise, clearly if both $|E_k|$ and $|F_k|$ are not greater than $\frac{\sqrt{105}-9}{6}$, and one is strictly less than that number, $\delta_{k+1} < \delta_k$

Q.E.D.

Using equations (3-19) and (3-21) of the proof, and calculating maximum values for $|E_o|$ and $|F_o|$ it is possible to calculate the maximum error for any iteration. This has been done for a few initial values and plotted on Fig. 3-3. Fig. 3-4 shows the number of significant bits generated after each iteration. Clearly, this algorithm converges very rapidly. Although it is not second order, it does increase the average number of places of accuracy of B_k and R_k by more than 50% per iteration.

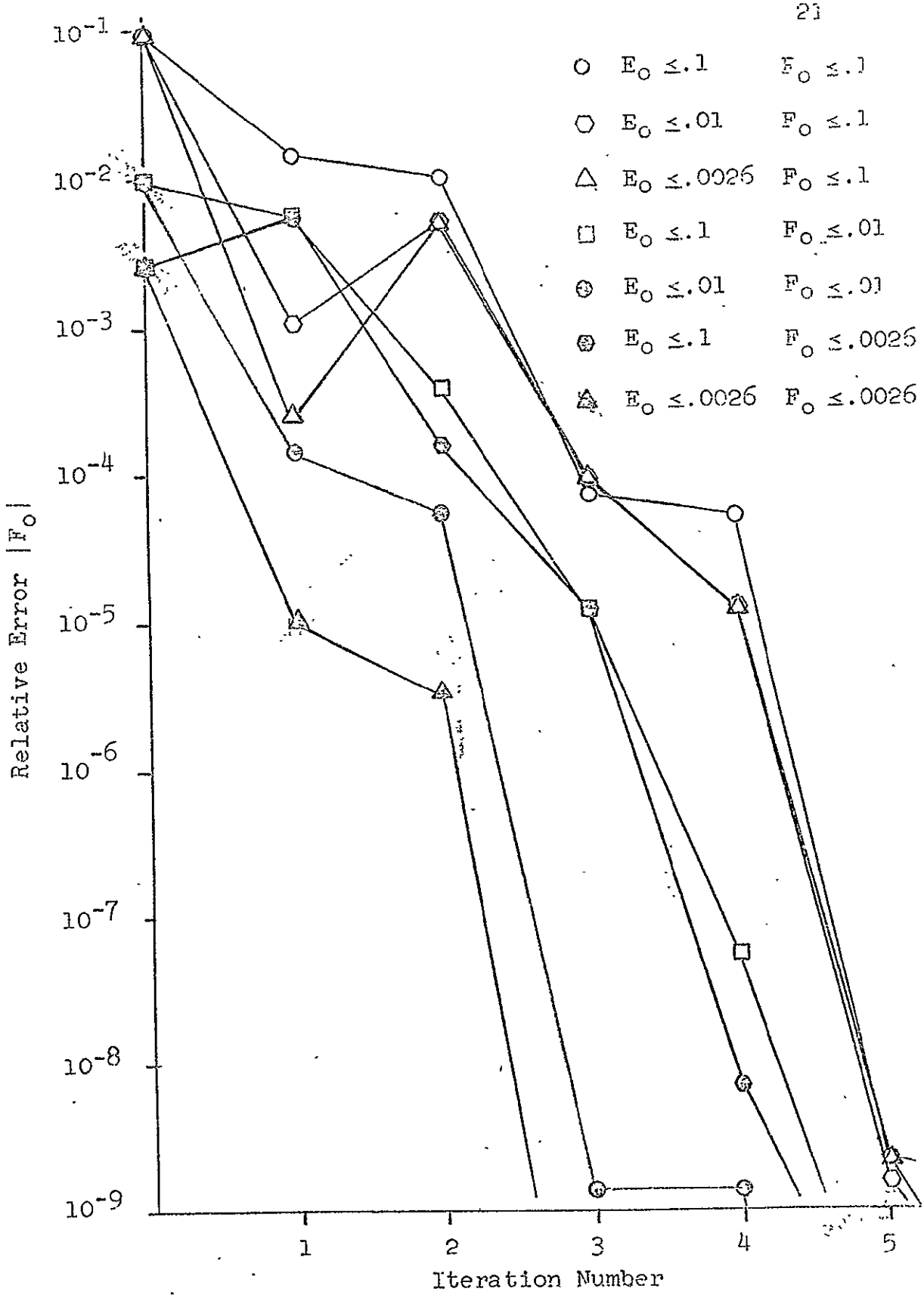


Fig. 3-3 Maximum Relative Error vs. Iteration Number for a Variety of Initial Conditions--Algorithm R.

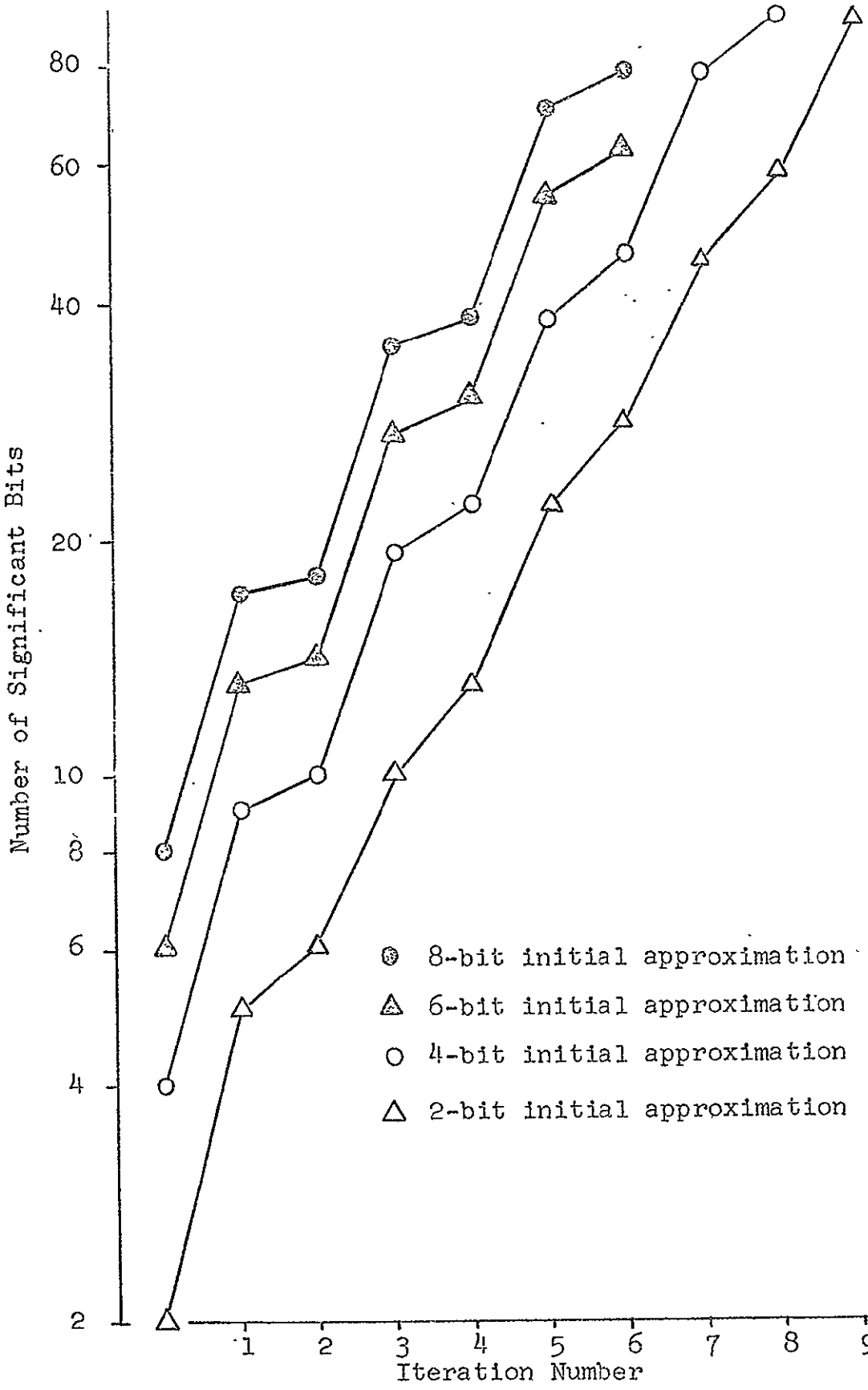


Fig. 3-4 Minimum Number of Significant Bits vs. Iteration Number Given Maximum Initial Errors B_o and R_o--Algorithm R.

CHAPTER 4

SIMULATION STUDIES OF ALGORITHM R

4.1 Method

In order to establish some of the characteristics of the nondividing algorithm R and compare it to other methods a simulation was performed on the CDC 6600. The algorithm was studied, varying the simulated word length, and initial approximation accuracy.

Word length variations were simulated by rounding the word after a fixed number of bits. After each multiplication the product was rounded after a prescribed number of bits. The same subroutine rounded initial approximations in order to study their effect on the convergence rate.

The numbers were assumed to be fixed point, since all numbers except N are between 1/2 and 2--a very narrow range. N will be normalized so that $1/4 \leq N < 1$. The assumption was made to speed up the addition, since fixed point addition is much faster than floating point.

The initial approximation was generated in the following way: The number N and the true square root of that number was determined, using the Fortran library function. The reciprocal of this number also was calculated

and these two numbers were rounded after P places to represent P-bit approximations to B and R respectively. This simulates a look-up table using P bits of N to generate P-place approximations B_o and R_o.

4.2 Comparison

In order to rate algorithm R, a conventional method was simulated, assuming a division algorithm of the type used in the IBM 360/91. [1] This machine uses recursive multiplication after an initial approximation from a look-up table. It was assumed a standard Newton-Raphson iteration using division is used in the conventional method. The criterion considered for comparison purposes was the number of multiplications required.

The IBM 360/91 contains an extremely fast multiply/divide unit which performs division in the following way: For fraction N/D, generate R_o, an approximate reciprocal to D, using combinational logic. Multiply both the numerator and denominator by R_o. The numerator product is now approximately equal to the quotient, while the denominator product is very close to unity. It can be shown [1] that taking the two's complement of the new denominator and again multiplying both numerator and denominator by this complement, the denominator will

approach unity at a second order rate, i.e., its number of consecutive ones or zeros will double with each iteration. This algorithm can be represented thus:

$$Q = \frac{N}{D} \cdot \frac{R_0}{R_0} \cdot \frac{R_1}{R_1} \cdot \frac{R_2}{R_2} \cdots \frac{R_N}{R_N} = \frac{Q}{1} \qquad (4\text{-}1)$$

where R_0 is a P-bit approximation to 1/D

and $$R_{k+1} = 2 - D \cdot \prod_{j=1}^{k} R_j \, . \qquad (4\text{-}2)$$

The criterion used for stopping both iterations was when $$|B_k - B| \leq 2\epsilon \qquad (4\text{-}3)$$

where ϵ is the value of the least significant bit of the simulated machine.

Initially, a large number of values for N were used. These numbers were evenly spaced between 1/4 and 1. Later a random number generator was used to generate an array of 1000 random numbers. No variations in convergence rate were detected with respect to the size of the number N. The first method, using evenly spaced values of N, had a somewhat wider variance, due to the fact that it hit some perfect squares, such as 1/4, which require no iterations. This, of course, did not happen often with random numbers and the variance was extremely small, usually less than 0.2 and in many cases less than 0.1.

The simulation was performed twice--once to find the average number of iterations required and once to find the variance. A flow chart for the simulation is shown in Fig. 4-1. The abbreviation I.A. is for initial approximation, i.e., an integer which represents the number of significant bits of B_o and R_o.

The comparative numbers of multiplications required (3 per iteration for algorithm R, 2 per iteration for the conventional algorithm) are plotted on Figs. 4-2, 4-3, 4-4, 4-5, and 4-6, for word lengths of 20, 32, 48, 64, and 92. In virtually all cases, algorithm R requires fewer multiplications. The greatest advantage is realized in the larger machines, which is particularly remarkable since the conventional algorithm is intended for large machines.

Another characteristic which becomes more apparent when the comparisons are plotted on linear paper is that the slope of the R algorithm is much smaller, i.e., not as much is lost by a less accurate initial approximation. This is shown in Fig. 4-7 for a 64-bit machine.

Also on graphs 4-2 to 4-6 is shown the required number of multiplications necessary to guarantee convergence from the initial approximation. Although for small machines it is greater than the conventional algorithm,

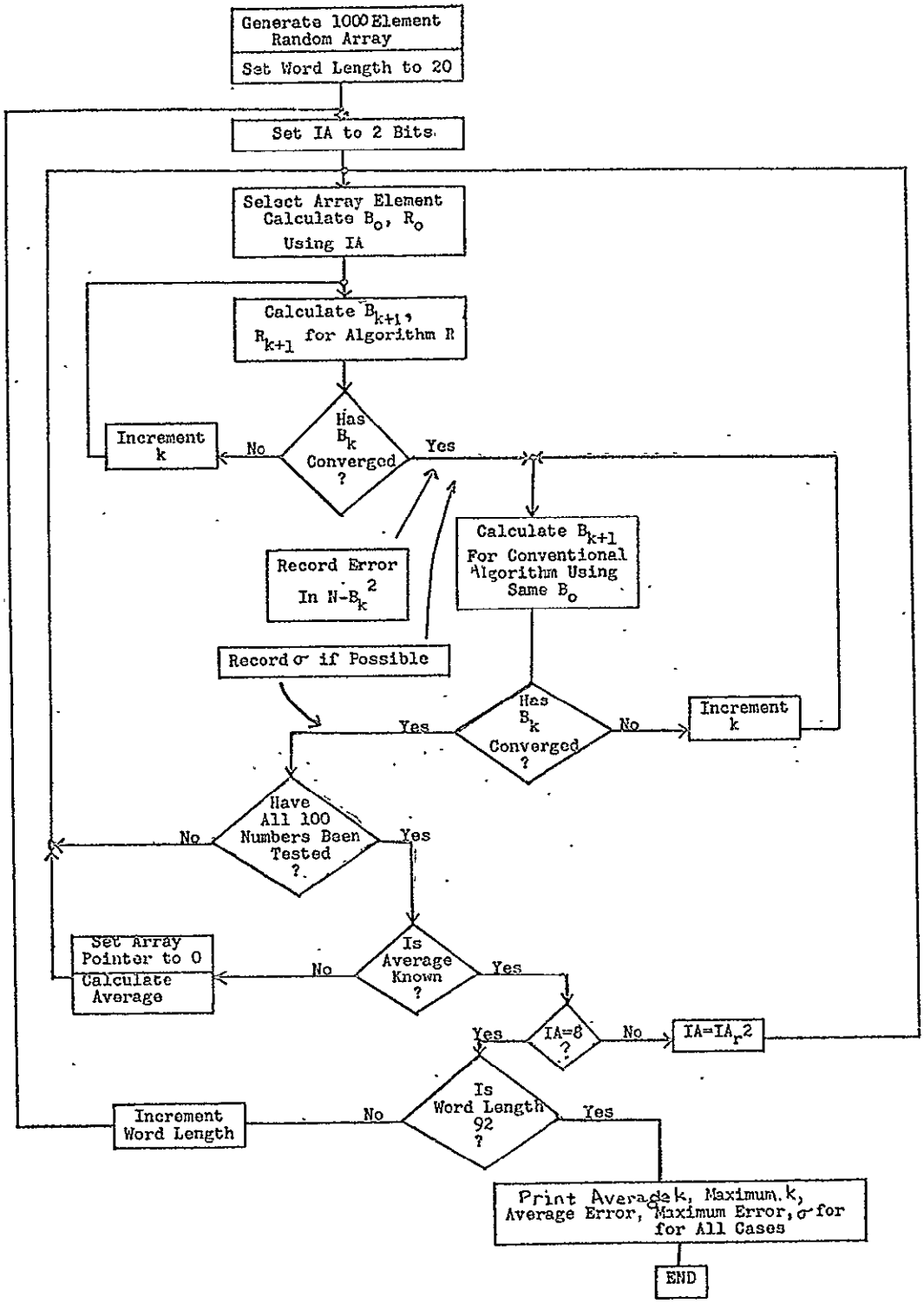


Fig. 4-1 Flow Chart of Simulation of Algorithm R

△ Average for simulation using Conventional Algorithm

○ Average for simulation using Algorithm R

◎ Maximum required to guarantee convergence

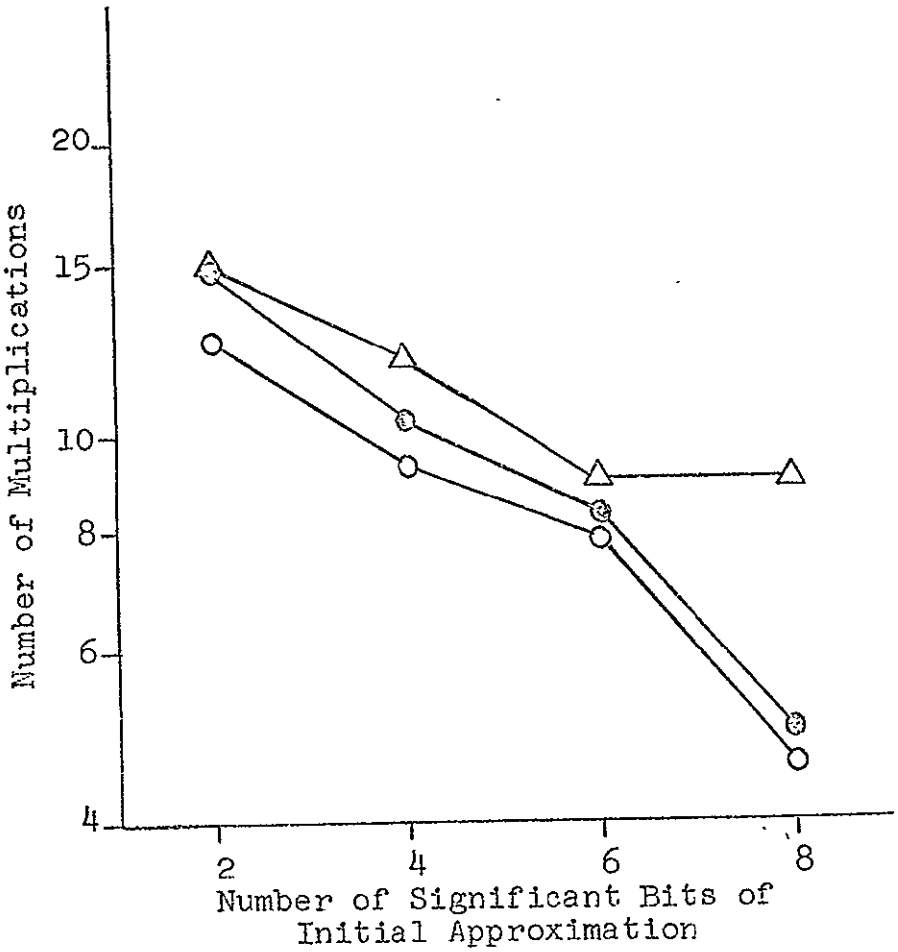


Fig. 4-2 Number of Multiplications for a 20-bit Number vs. Accuracy of Initial Approximation.

△ Average for simulation using Conventional Algorithm

○ Average for simulation using Algorithm R

⊛ Maximum required to guarantee convergence

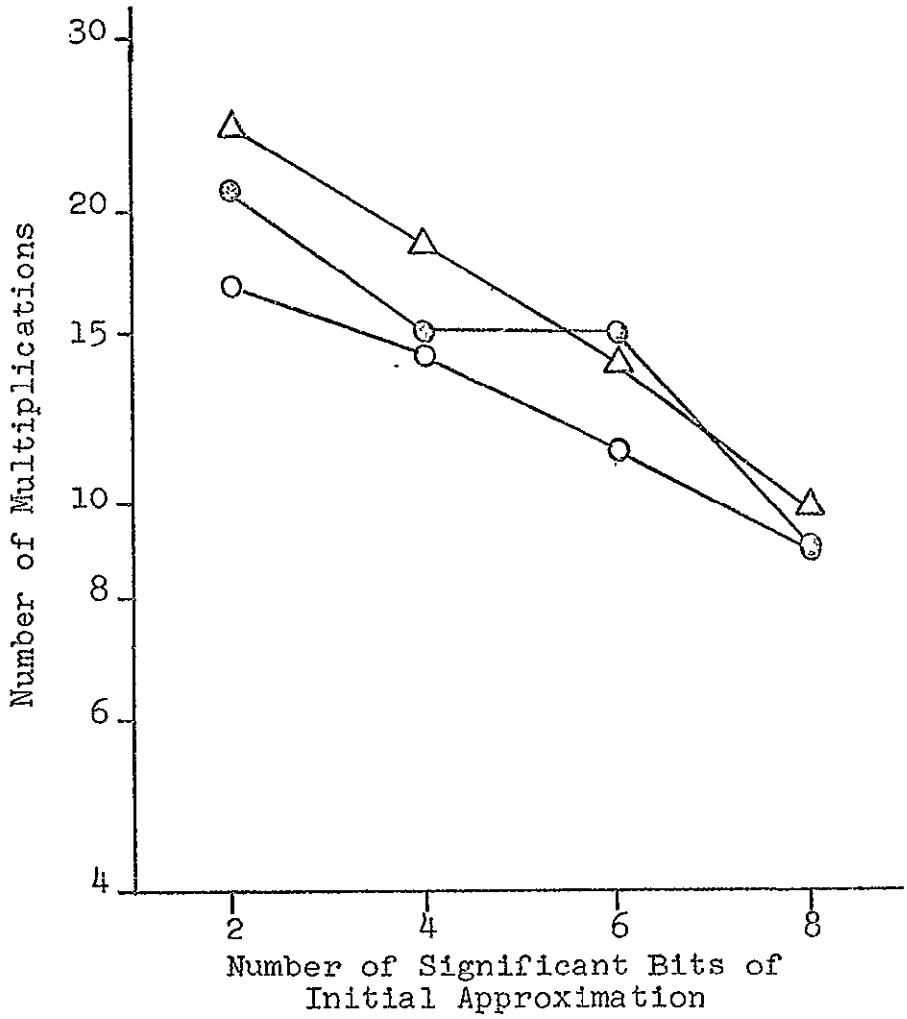


Fig. 4-3 Number of Multiplications for a 32-bit Number vs. Accuracy of Initial Approximation.

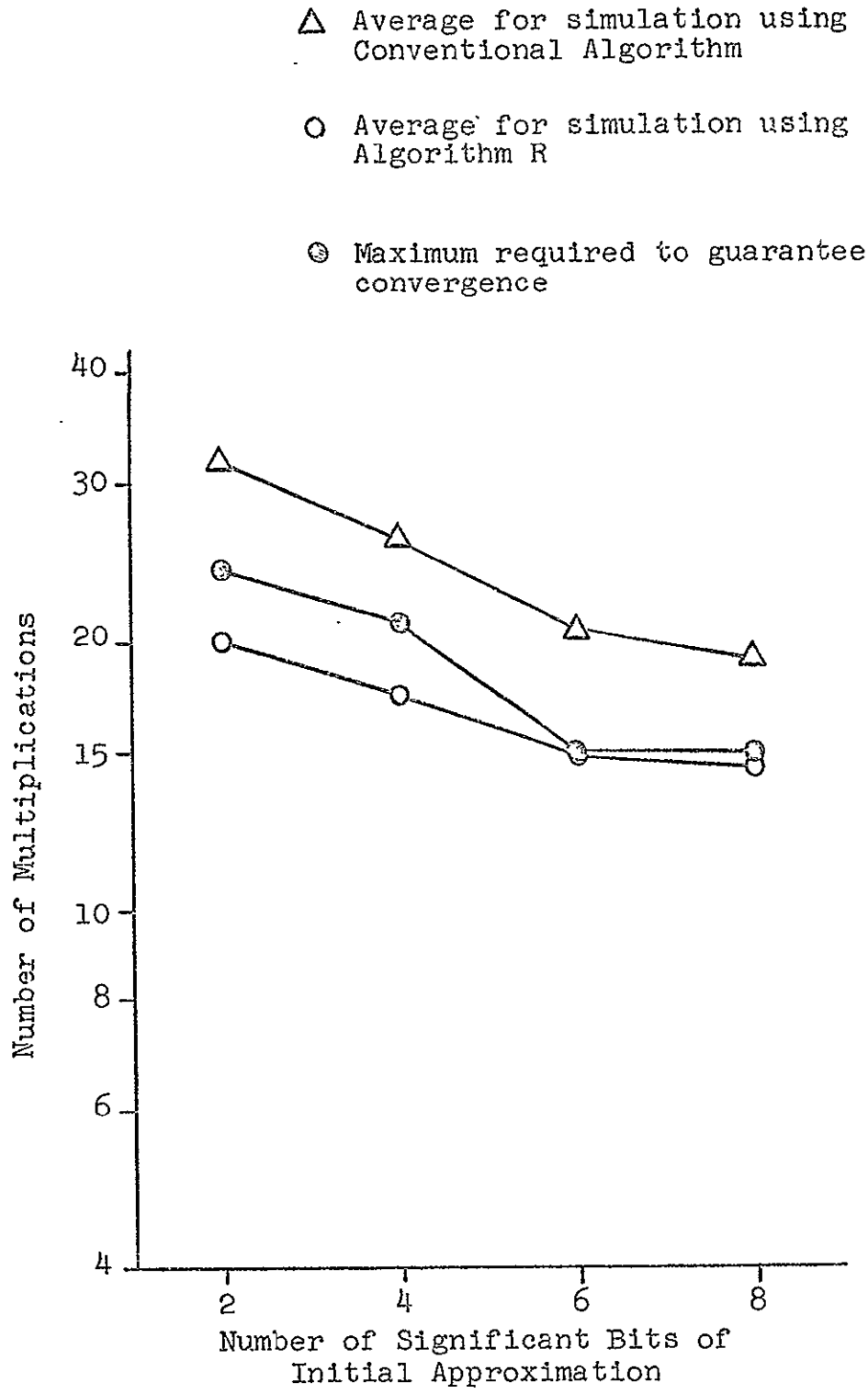


Fig. 4-4 Number of Multiplications for a 48-bit Number vs. Accuracy of Initial Approximation.

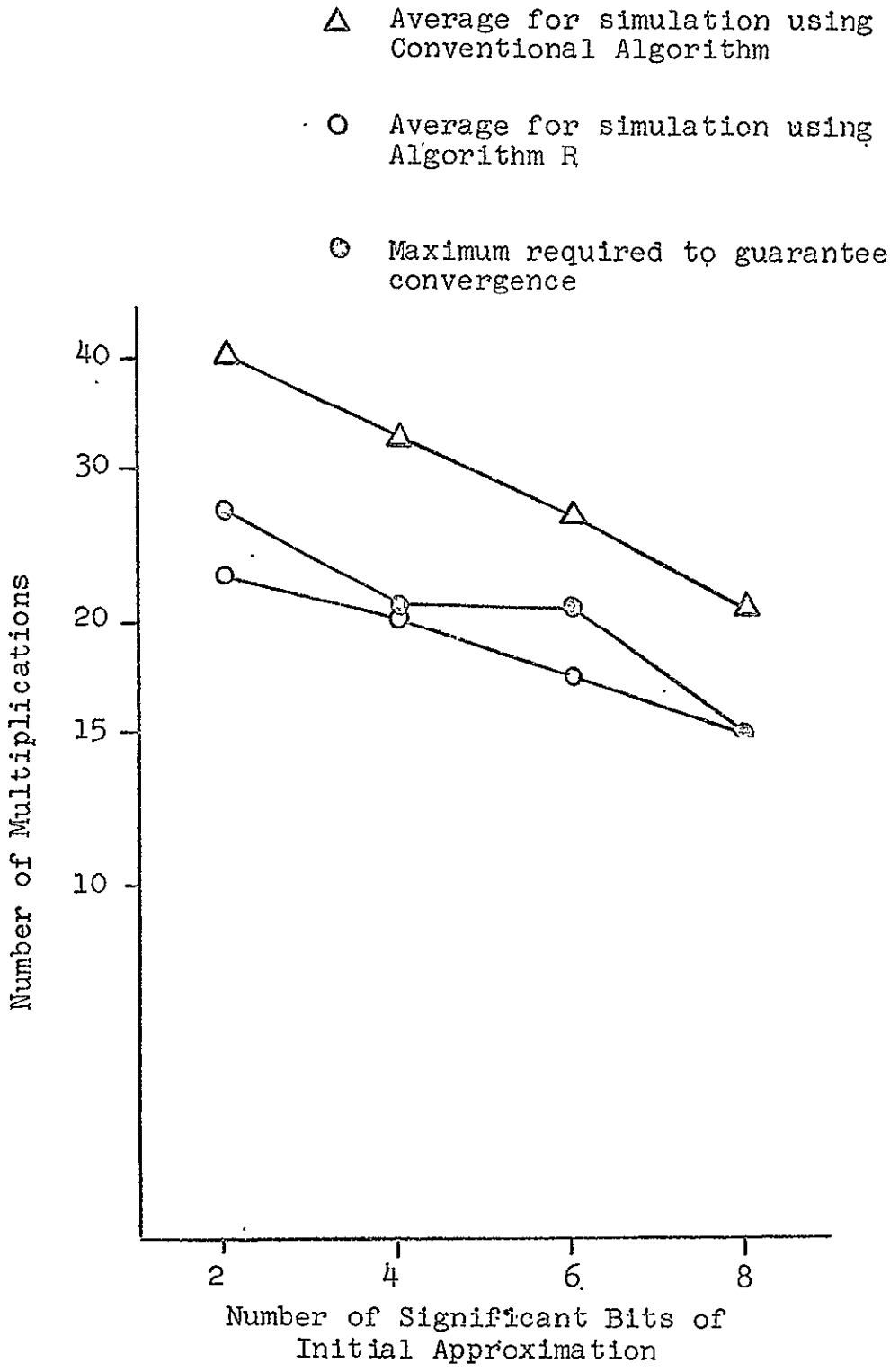


Fig. 4-5 Number of Multiplications for a 64-bit Number vs. Accuracy of Initial Approximation.

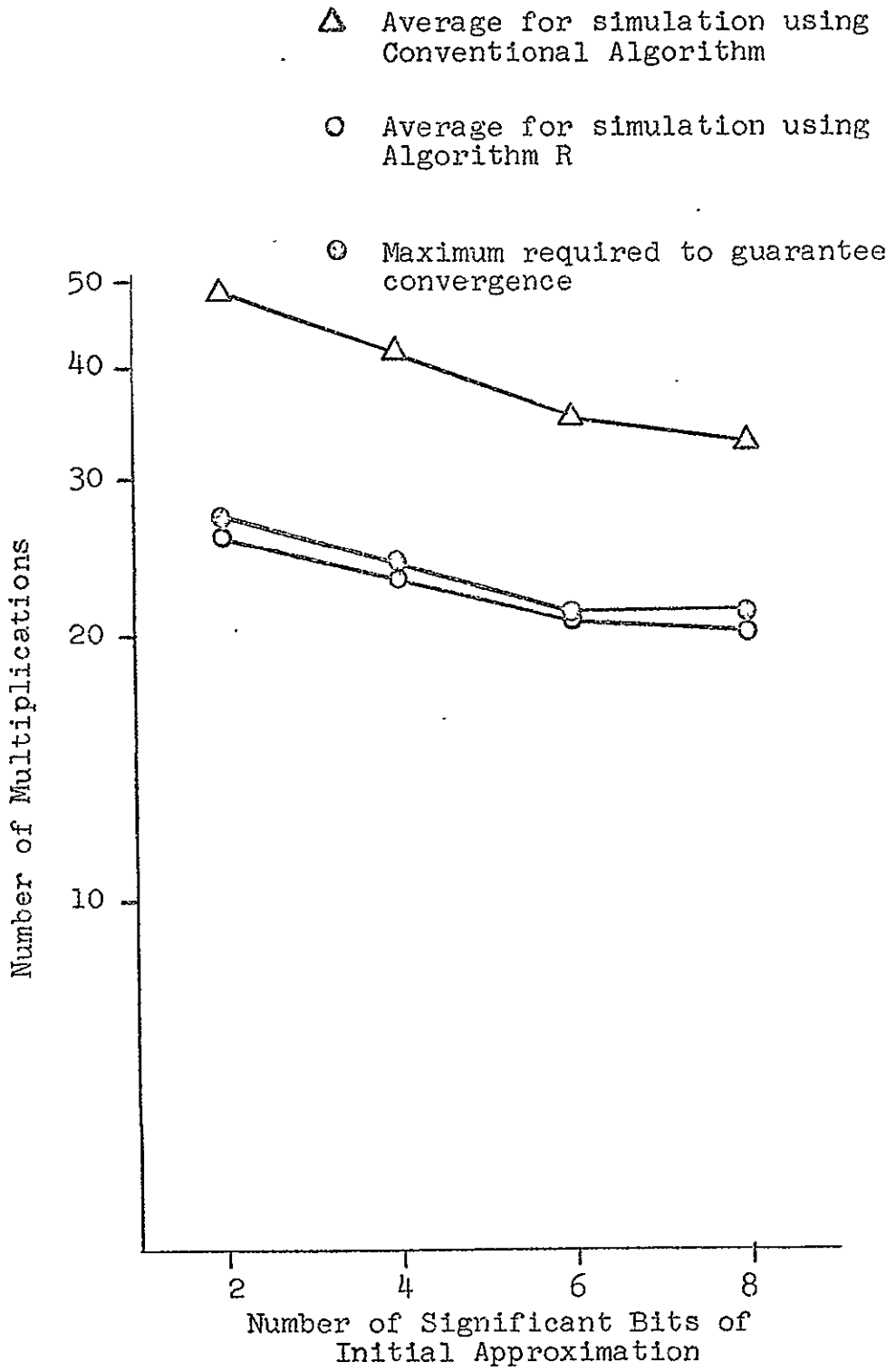


Fig. 4-6 Number of Multiplications for a 92-bit Number vs. Accuracy of Initial Approximation.

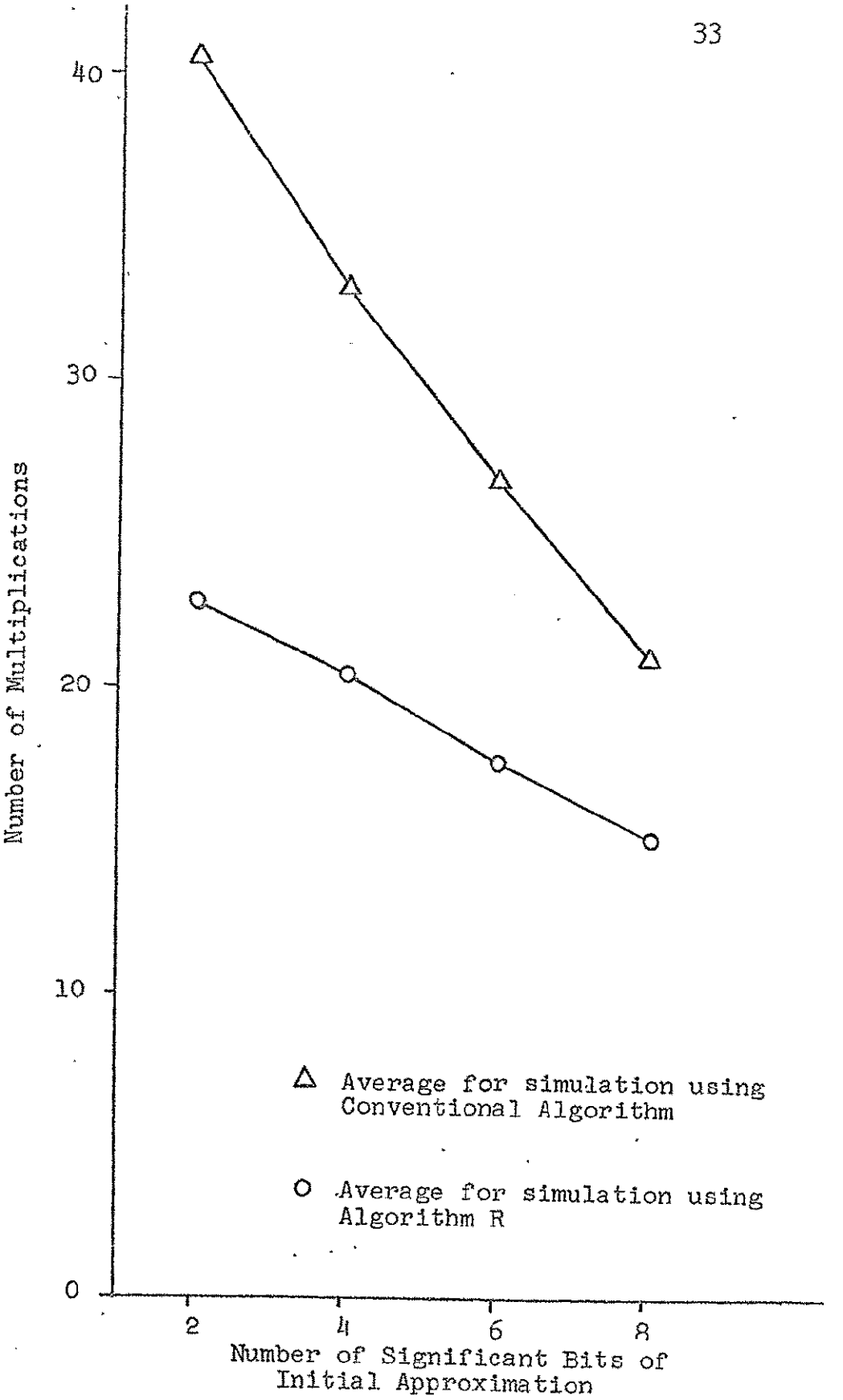


Fig. 4-7 Number of Multiplications for a 64-bit Number vs. Accuracy of Initial Approximation.

it is much smaller on the large machines, barely greater than the average number.

4.3 Conclusion

Algorithm R converges very rapidly, with its most impressive advantage in a large machine. It does not require an extremely accurate initial approximation to converge rapidly.

CHAPTER 5

ALGORITHM F

5.1 Motivation

In the simulation of algorithm R a comparison was made with the Newton-Raphson square root method employing the IBM 360/91 division scheme. In this machine a high speed look-up table was available, implemented in combinational logic. Algorithm R assumed a similar look-up table to generate initial values R_o and B_o, although it was shown that a much smaller table could be used.

The question arises, would it be possible to use the 360/91 look-up table or a similar one more effectively to generate square root? Thus algorithm F was developed, assuming that an approximate division could be performed. The algorithm may be stated in the following way: For normalized number $N = B^2$, $1/4 \leq N < 1$, $B_o \cong B$, the iteration

$$B_{k+1} = B_k + f_p(B_k) \cdot (N - B_k^2) \qquad (5\text{-}1)$$

where $f_p(B_k)$ is one-half the reciprocal of B_k rounded to p bits. This requires the 360/91 reciprocal generator with an output shifting it right one place. The justification for Algorithm F is this: The Newton-Raphson iteration may be written

$$B_{k+1} = \frac{1}{2}\left(B_k + \frac{N}{B_k}\right) \tag{5-2}$$

$$= \frac{1}{2}\left(2B_k + \frac{N}{B_k} - B_k\right) \tag{5-3}$$

$$= B_k + \frac{1}{2}\left(\frac{N}{B_k} - \frac{B_k^2}{B_k}\right) \tag{5-4}$$

$$= B_k + \frac{1}{2B_k}(N - B_k^2) \tag{5-5}$$

The term $\frac{1}{2B_k}$ is essentially a weighting factor for the correction term $N-B_k^2$. Conceivably, as in algorithm R, it could be iterated to greater accuracy, also. However, algorithm F converges very rapidly, using only the rounded approximation. The proof, given later, shows that if the value of B_k has at least as many significant bits as the reciprocal generator, i.e., p+1 bits, that the convergence rate will be at least p bits per iteration. For specific cases, where the maximum initial error is given, it is possible to calculate the maximum possible error after each iteration which demonstrates an even greater convergence rate.

A sequence chart for F is shown in Fig. 5-1 Two multiplication and two additions are required per iteration. However, since one factor in each multiplication, B_k in the first and $f_p(B_k)$ in the second, are already normalized, no normalization is necessary either in the

multiplication or in the addition. Thus the time required is approximately the time needed for two multiplications.

An example of algorithm F is given in Fig. 5-2 for algorithm R in Fig. 3-2. A 6-bit reciprocal generator (p = 6) was assumed. The example converges at approximately the same rate as for algorithm R under these conditions.

5.2 Proof of Convergence

Theorem F: For a number $N=B^2$, $1/4 \le N < 1$, given an initial approximation $1/2 \le B_o < 1$ and the iteration

$$B_{k+1} = B_k + f_p(B_k) \cdot (N - B_k^2) \qquad (5\text{-}6)$$

where $f_p(x)$ is any function such that

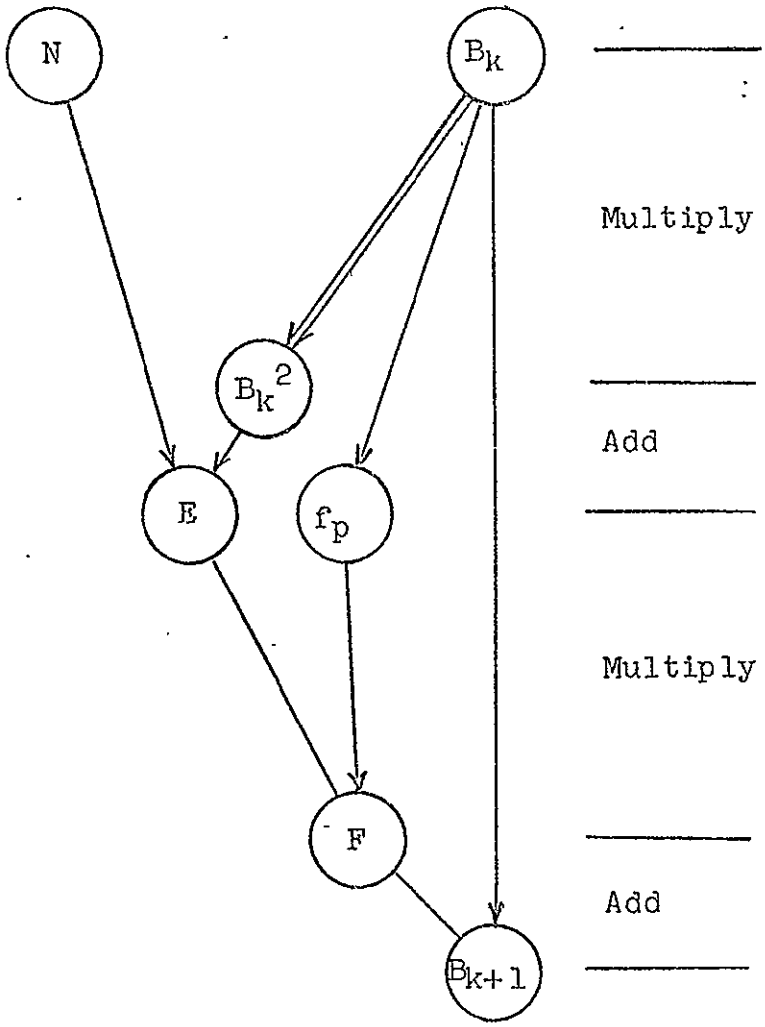
$$\frac{1}{2}\left(\frac{1}{x} - 2^{-p-1}\right) \le f_p(x) \le \frac{1}{2}\left(\frac{1}{x} + 2^{-p-1}\right) \qquad (5\text{-}7)$$

then $\lim_{k \to \infty} B_k = B$. Furthermore, if

$$0 < B_k - B \le 2^{-p-1}, \quad \left|\frac{B_{k+1}-B}{B_k-B}\right| \le 2^{-p} \qquad (5\text{-}8)$$

Proof: Define again relative error function

$$F_k = \frac{B_k - B}{B} \qquad (5\text{-}9)$$



$$B_{k+1} = B_k + f_p \cdot (N - B_k^2)$$

Fig. 5-1 Sequence Diagram for Algorithm F.

N=.45313

B(0)= .6875000000000	F(0)= 2.13E-02
B(1)= .6734655078125	F(1)= 4.70D-04
B(2)= .6731528181247	F(2)= 5.20E-06
B(3)= .6731493544089	F(3)= 5.89E-08
B(4)= .6731493152365	F(4)= 6.66E-10
B(5)= .6731493147934	F(5)= 7.53E-12
B(6)= .6731493147884	F(6)= 8.44E-14
B(7)= .6731493147883	F(7)=-5.28E-15

B(T)= .6731493147883

Fig. 5-2 Example of Algorithm F.

Then

$$F_{k+1} = \frac{B_k + f_p(B_k)[N-B(F_k+1)^2] - B}{B} \qquad (5\text{-}10)$$

Substituting $B^2 = N$ and $F_k = B_k - B$

$$F_{k+1} = F_k - f_p(B_k) \cdot F_k \cdot (2+F_k) \cdot B \qquad (5\text{-}11)$$

From hypothesis we can see that either

$$|F_{k+1}| \leq |F_k - \frac{1}{2}\left(\frac{1}{B_k} - 2^{-p-1}\right) F_k (2 - F_k) B| \qquad (5\text{-}12)$$

or

$$|F_{k+1}| \leq |F_k - \frac{1}{2}\left(\frac{1}{B_k} + 2^{-p-1}\right) F_k (2 - F_k) B| \qquad (5\text{-}13)$$

(5-12) and (5-13) may be combined as

$$|F_{k+1}| \leq |F_k - \frac{1}{2B_k}(F_k)(2-F_k)B| + |F_k - 2^{-p-2}(F_k)(2-F_k)B| \qquad (5\text{-}14)$$

Substituting for B_k and rearranging gives

$$|F_{k+1}| \leq |F_k - F_k\frac{2+F_k}{2+2F_k}| + |F_k(2B+F_k)2^{-p-2}| \qquad (5\text{-}15)$$

$$\left|\frac{F_{k+1}}{F_k}\right| \leq \left|1 - \frac{2B+2F_k-F_k}{2B+2F_k}\right| + \left|(2B+F_k)2^{-p-2}\right| \qquad (5\text{-}16)$$

$$\left|\frac{F_{k+1}}{F_k}\right| \leq \left|\frac{F_k}{2(1+F_k)}\right| + \left|(2B+F_k)B \cdot 2^{-p-2}\right| \qquad (5\text{-}17)$$

From (2.12) we know

$$\left|\frac{F_{k+1}}{F_k}\right| \leq \frac{1}{2}\left|\frac{F_k}{1+F_k}\right| + 2^{-p-1} + |F_k|\ 2^{-p-2} \qquad (5\text{-}18)$$

Now if $1/2 \leq B_k \leq 1$, $1/2 \leq B \leq 1$

$$-1/2 \leq F_k = \frac{B_k}{B} - 1 < 1 \qquad (5\text{-}19)$$

Consider the function $F_k/(1+F_k)$ over the domain $-1/2 \leq F_k \leq 1$. It is a monotonically increasing function with extremes of -1 and +1/2 at $F_k = -1/2$ and $F_k = 1$ respectively. Thus

$$\frac{1}{2}\left|\frac{F_k}{F_{k+1}}\right| \leq \frac{1}{2} \qquad (5\text{-}20)$$

Hence from (5-18)

$$\left|\frac{F_{k+1}}{F_k}\right| \leq \frac{1}{2} + \frac{1}{2} \cdot 2^{-p} + \frac{1}{4} \cdot 2^{-p} \qquad (5\text{-}21)$$

If $P \geq 1$ this ratio is less than one and the series converges.

Case 1: $0 < F_k$

Clearly all terms of right side (5-17) are positive, so the absolute values can be deleted from the right side:

$$\left|\frac{F_{k+1}}{F_k}\right| \leq \frac{F_k}{2(1+F_k)} + B \cdot 2^{-p-1} + B \cdot F_k \cdot 2^{-p-2} \tag{5-22}$$

Expanding the first term in a Maclaurin series

$$\frac{F_k}{2(1+F_k)} = \frac{1}{2} F_k - \frac{1}{2} F_k^2 + \frac{1}{2} F_k^3 - \cdots \tag{5-23}$$

$$= \frac{1}{2} F_k - \frac{1}{2} F_k \frac{F_k}{2(1+F_k)} \tag{5-24}$$

$$\left|\frac{F_{k+1}}{F_k}\right| \leq \frac{1}{2} F_k + B \cdot 2^{-p-1} + B \cdot F_k \cdot 2^{-p-2} - \frac{1}{2} F_k \frac{F_k}{1+F_k} \tag{5-25}$$

$$0 < F_k = \frac{B_k - B}{B} \leq \frac{2^{-p-1}}{B} < 2 \tag{5-26}$$

Since the last term is negative, it may be dropped.

$$\left|\frac{F_{k+1}}{F_k}\right| \leq \frac{1}{B} 2^{-p-2} + B \cdot 2^{-p-1} + 2^{-2p} \tag{5-27}$$

$$\left|\frac{F_{k+1}}{F_k}\right| \leq 2^{-p-1}\left(\frac{1}{2B} + B + 2^{-p+1}\right) \tag{5-28}$$

Now consider the function

$$g(B) = \frac{1}{2B} + B + C \tag{5-29}$$

where C is a constant.

Its maximum value on the interval $\frac{1}{2} \leq B \leq 1$ is at B=1 where $g(B) = \frac{3}{2} + C$ (5-30)

Substituting this value in (5-24) gives

$$\left|\frac{F_{k+1}}{F_k}\right| \leq 2^{-p-1} \left(\frac{3}{2} + 2^{-p+1}\right) \quad (5\text{-}31)$$

If $p \geq 3$

$$\frac{3}{2} + 2^{-p+1} \leq \frac{3}{2} + \frac{1}{4} \leq 2 \quad (5\text{-}32)$$

Therefore, for $F_k > 0$

$$\left|\frac{F_{k+1}}{F_k}\right| < 2^{-p} \quad (5\text{-}33)$$

Case 2: $F_k < 0$

The first term on the right hand side of (5.17) is negative while the second term is positive.

$$\left|\frac{F_{k+1}}{F_k}\right| \leq -\frac{F_k}{2(1+F_k)} + B^{-p-1} + B \cdot F_k \cdot 2^{-p-2} \quad (5\text{-}34)$$

Again, expanding the first term gives

$$\left|\frac{F_{k+1}}{F_k}\right| < -\frac{1}{2} F_k + B \cdot 2^{-p-1} + B \cdot F_k \cdot 2^{-p-2} + \frac{1}{2} F_k \frac{F_k}{1+F_k} \quad (5\text{-}35)$$

Dropping the third term of (5-35) and noting that

$$-F_k < \frac{2^{-p-1}}{B} \tag{5-35}$$

and

$$\frac{1}{2} F_k \frac{F_k}{1+F_k} < F_k^2 \tag{5-36}$$

gives

$$\left| \frac{F_{k+1}}{F_k} \right| \leq \frac{2^{-p-2}}{B} + B \cdot 2^{-p-1} + 2^{-p\ 1} \tag{5-37}$$

This is equation (5-24) so that we may conclude that for $0 < |F_k| < \frac{2^{-p-1}}{B}$ (5-38)

$$\left| \frac{F_{k+1}}{F_k} \right| < 2^{-p} \qquad p \geq 3 \tag{5-39}$$

Q.E.D.

Using equation (5-17) and assuming an initial error $F_o \leq 1$, the maximum error after each iteration was calculated for four values of p. The results are plotted in Fig. 5-3.

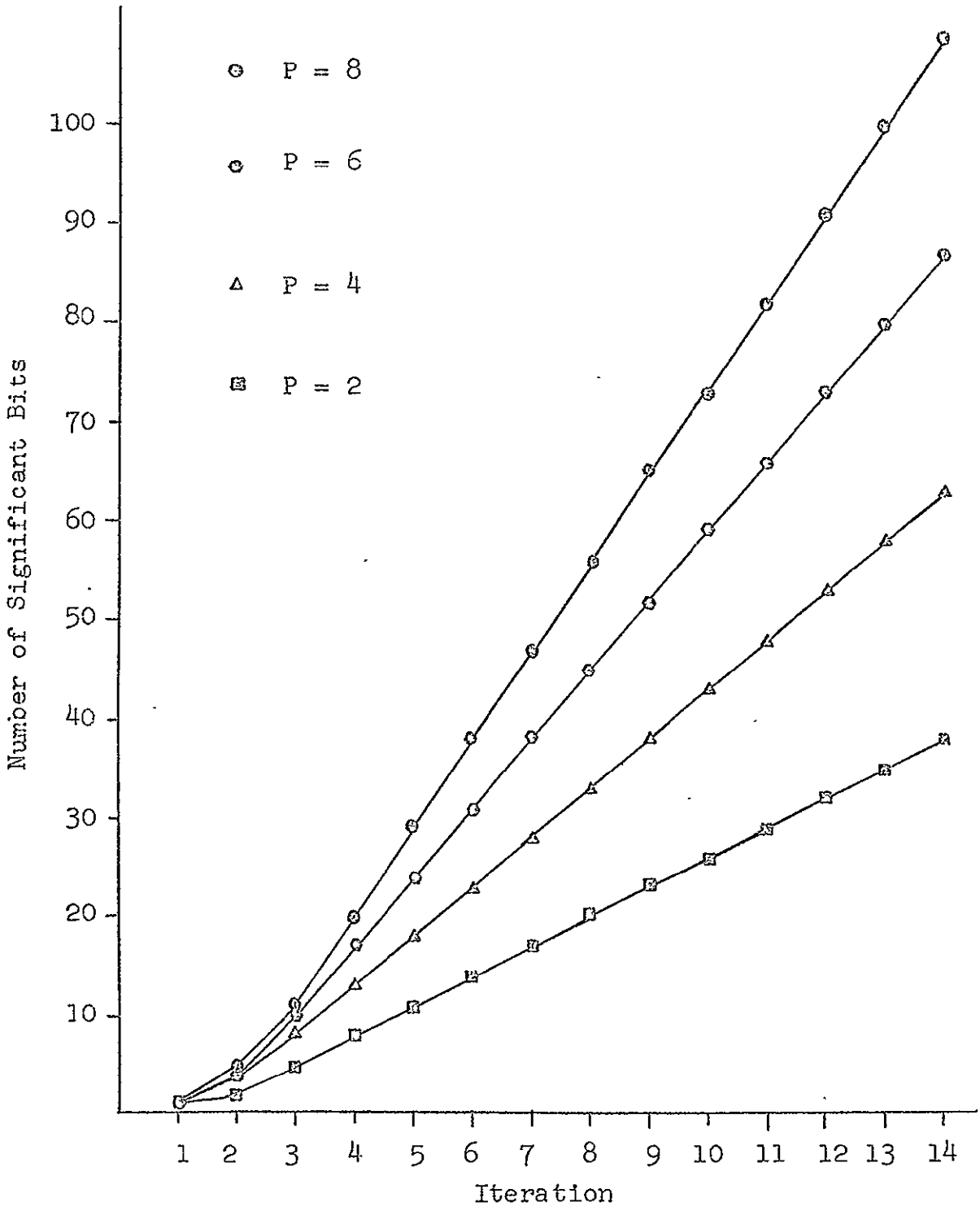


Fig. 5-3 Minimum Number of Significant Bits vs. Iteration Number with Initial Relative Error $|F_o| \leq 1$ for Four Values of Reciprocal Generator Accuracy.

CHAPTER 6

SIMULATION STUDIES OF ALGORITHM F

6.1 Method

A simulation of algorithm F was performed using virtually the same procedure as the simulation of algorithm R. A major change was made in that a new initial approximation method was used.

Since algorithm F is greatly enhanced by an accurate initial value, it was desired to generate an accurate initial value without the necessity of a second look-up table, i.e., utilize the reciprocal generator already available. Numerous methods were considered and one method which is quite satisfactory, if not optimal, was selected. It is adapted from a software initial approximation developed by Maehly [5] known as the best-fit method.

Maehly's algorithm is of the form

$$B_0 = \alpha + \frac{\beta}{\gamma+N} \qquad (6\text{-}1)$$

for constants $\alpha = 2.185183$, $\beta = 3.022900$, $\gamma = 1.545158$. In this form it guarantees a relative error of less than 2.6×10^{-3}, i.e., nine significant bits. The adaptation is to assume α, β, and γ are wired constants and use the reciprocal generator to generate $\frac{1}{\gamma+N}$. If γ and α have P

significant bits and β has P/2, then B_o will have P significant bits if $P \leq 9$.

This was simulated in the algorithm by rounding α and γ to P places, β to P/2 places. The result B_o then was also rounded to P places.

This method required two addition and one multiplication times, but it was felt that it is probably justified unless a look-up table for square root is to be implemented. Quite possibly an improved algorithm could give as great or greater accuracy faster or with fewer constants required, but this is considered out of the realm of this paper. This adaptation is sufficient for the simulation.

Another change in the simulation resulted in the fact that, by its nature, algorithm F has an easy test for convergence. The term $E = f_p(B_k) \cdot (N-B_k^2)$ is the correction factor, and as B_k converges it approaches zero. An easy test for convergence is to test E. $|E|$ less than ϵ guarantees that $|B_k-B| < 2\epsilon$. This can be shown as follows:

$$|E| = |f_p(B_k) \cdot (N-B_k^2)| < \epsilon \qquad (6\text{-}2)$$

$$B_{k+1} = B_k + E \qquad (6\text{-}3)$$

$$E = B_{k+1} - B_k = B\,(F_{k+1} - F_k) \qquad (6\text{-}4)$$

$$|F_{k+1} - F_k| = \frac{|E|}{B} < \frac{\epsilon}{B} \qquad (6\text{-}5)$$

Now if F_{k+1} and F_k have opposite signs then

$$|F_{k+1}| + |F_k| < \frac{\epsilon}{B} \tag{6-6}$$

$$|F_k| < \frac{\epsilon}{B} \tag{6-7}$$

$$|B_{k+1} - B| < \epsilon \tag{6-8}$$

If, however, F_{k+1} and F_k have the same sign then since from (5.17) we know if F_k is small compared to 2^{-p},

$$|F_{k+1}| \leq B^2 2^{-p} |F_k| \tag{6-9}$$

If $F_k \geq 0$ we know

$$F_{k+1} < B^2 2^{-p} F_k \tag{6-10}$$

$$\frac{\epsilon}{B} > F_k - F_{k+1} > F_k (1 - B^2 2^{-p}) \geq \frac{1}{2} F_k. \tag{6-11}$$

If $P \geq 1$

$$0 \leq F_k \leq \frac{2\epsilon}{B} \tag{6-12}$$

$$B_k - B < 2\epsilon . \tag{6-13}$$

If $F_k \leq 0$ likewise

$$F_{k+1} > B^2 2^{-p} F_k \tag{6-14}$$

$$\frac{\epsilon}{B} \quad F_{k+1} - F_k > F_k (B^2 2^{-p} - 1) > - \frac{1}{2} F_k. \tag{6-15}$$

If $P \geq 1$

$$0 \geq F_k \geq -\frac{2\epsilon}{B} \tag{6-16}$$

$$0 \geq B_k - B > -2\epsilon \tag{6-17}$$

$$|B_k - B| < 2\epsilon$$

For algorithm F, this test was used, where ϵ was the value of the least significant bit. One result of this is that for all simulations, one extra iteration was required, i.e., F_k was tested on iteration k+1 for convergence. As a result, in some case the maximum required number of iterations guaranteed by the initial approximation was actually exceeded by one iteration. In fact, for cases where the algorithm converged very fast, e.g., when the reciprocal generator produced a large number of significant bits, the <u>average</u> number of iterations made was greater than the maximum required number.

The same test was made for the conventional algorithm for convergence, i.e., stop if $|B_{k+1} - B| < \epsilon$. This resulted in a slightly higher value for this algorithm, also, than in the comparison with R.

The flow chart for the algorithm F simulation is shown in Fig. 6-1. Only random numbers were used in this simulation. The initial array size was 1000, but was

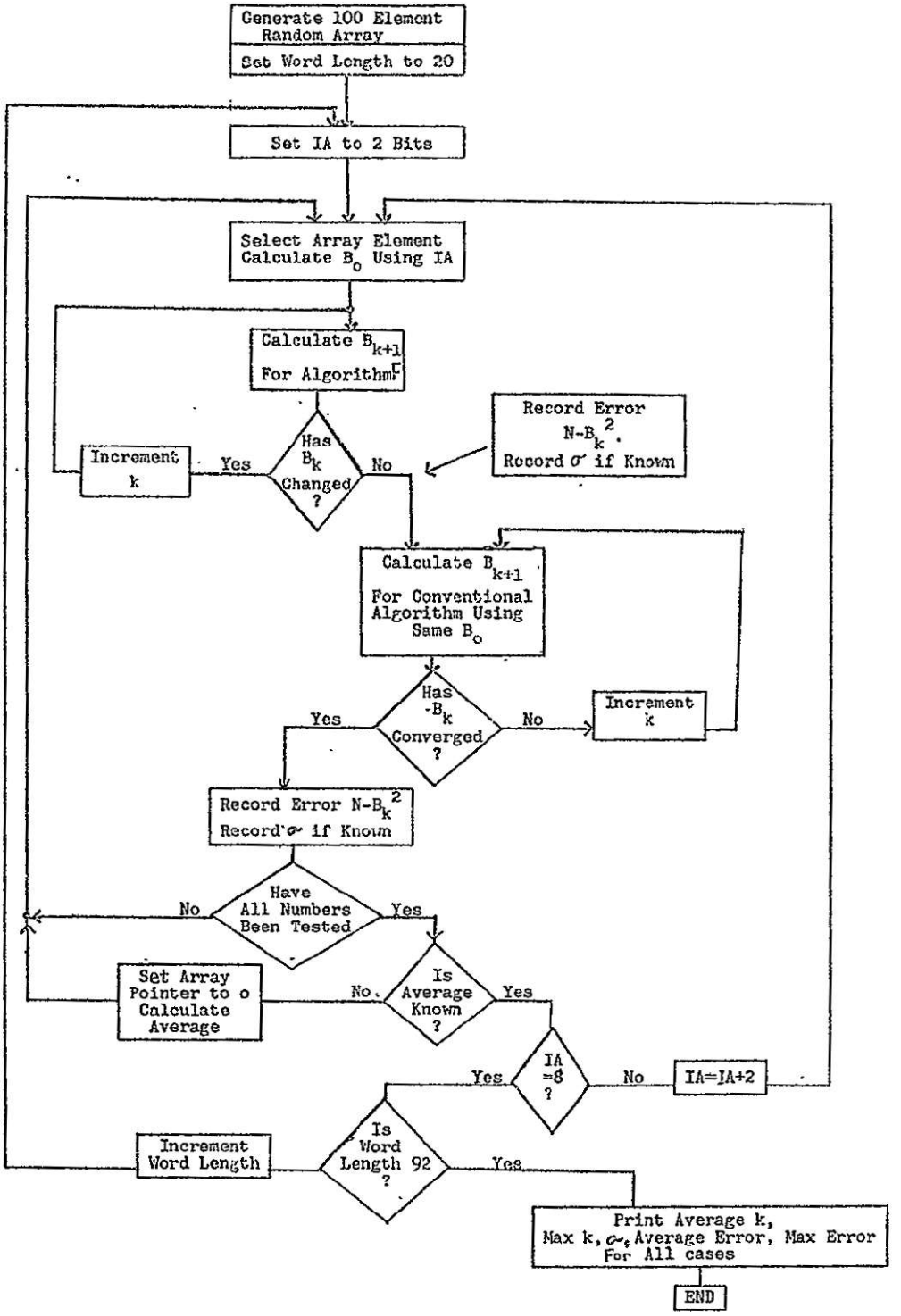


Fig. 6-1 Flow Chart for Simulation of Algorithm F

reduced to 100 after early results indicated little variation in data for the two sizes. The results of the simulation for word length of 20, 32, 48, 64, and 92 bits are shown in Figs. 6-2 through 6-6. Included are the data showing the number of iterations after which convergence is assured.

6.2 Conclusions

Algorithm F converges very rapidly--requiring in some cases, as little as 40% of the number of multiplications as the conventional algorithm. However, it is even more dependent on the initial approximation than the conventional algorithm, particularly for guaranteed convergence.

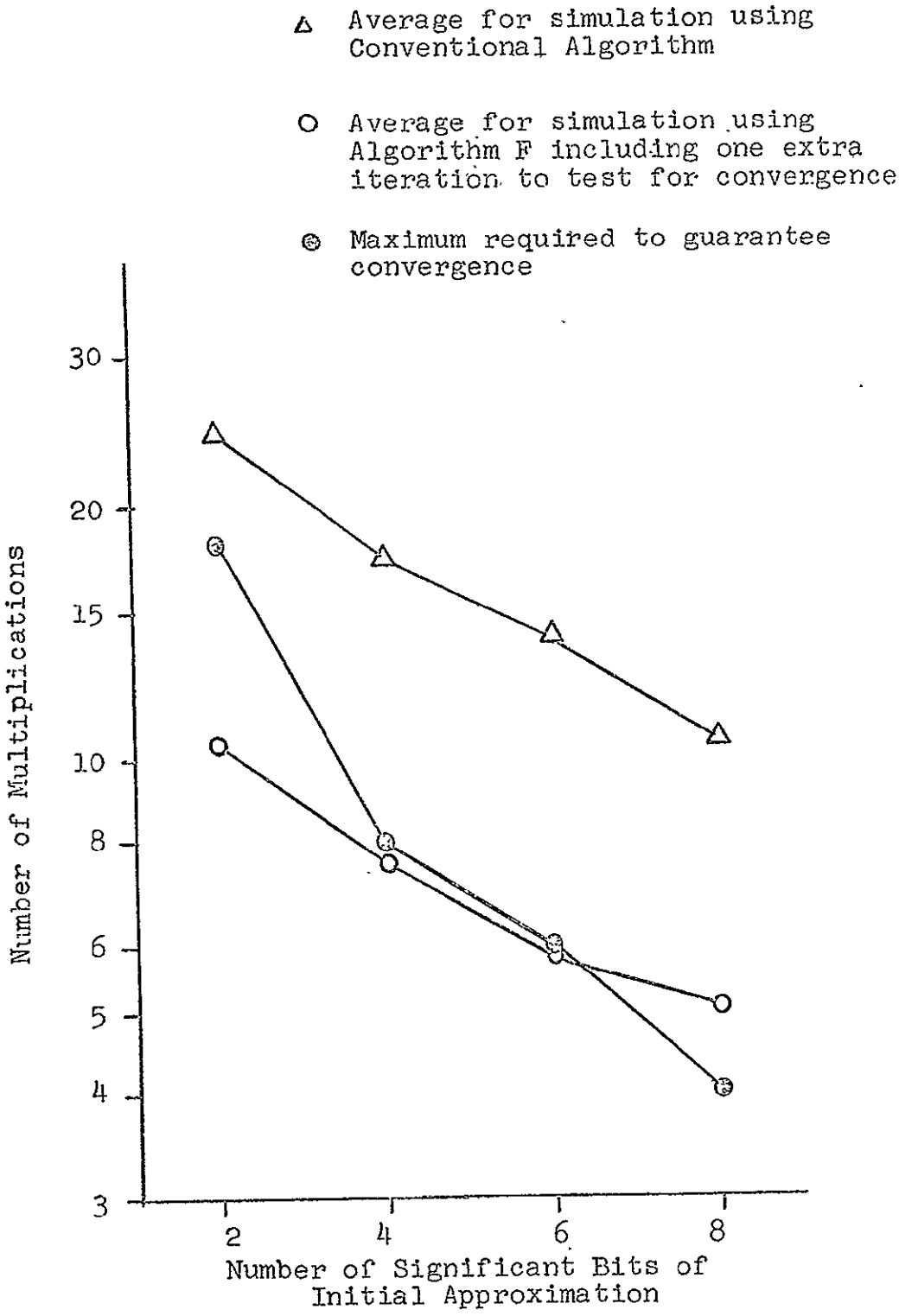


Fig. 6-2 Number of Multiplications for a 20-bit Number vs. Accuracy of Initial Approximation.

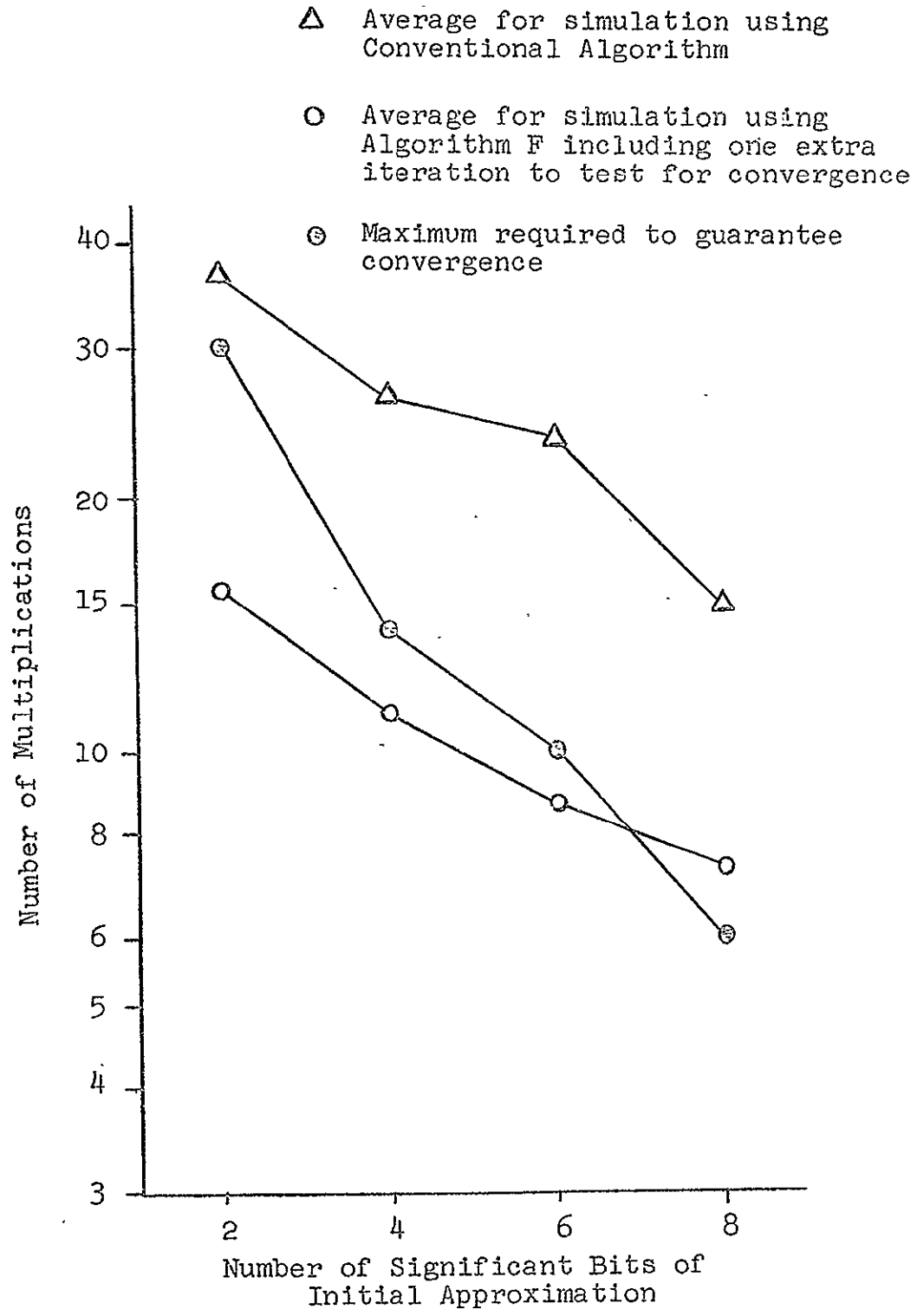


Fig. 6-3 Number of Multiplications for a 32-bit Number vs. Accuracy of Initial Approximation.

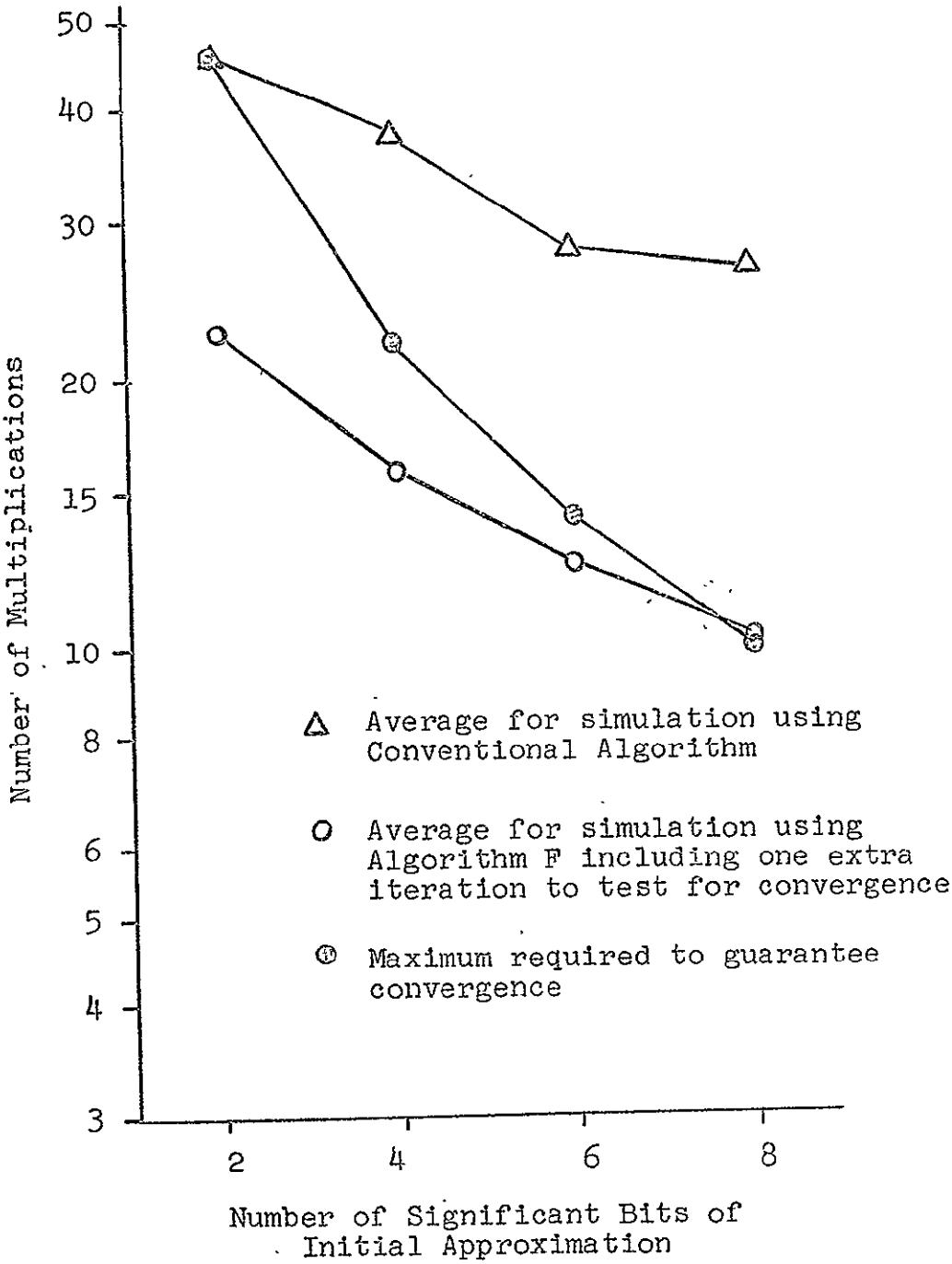


Fig. 6-4 Number of Multiplications for a 48-bit Number vs. Accuracy of Initial Approximation.

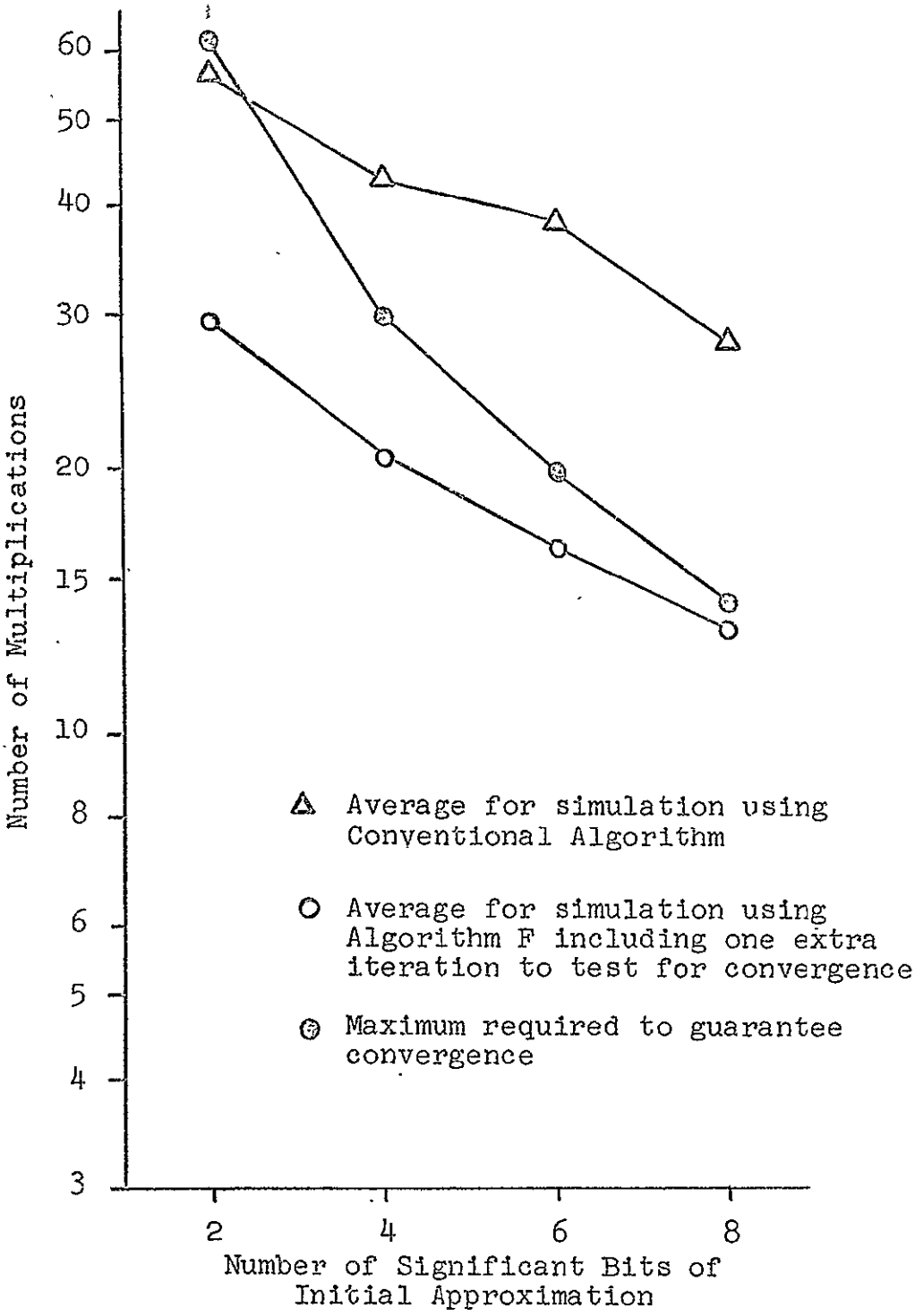


Fig. 6-5 Number of Multiplications for a 64-bit Number vs. Accuracy of Initial Approximation.

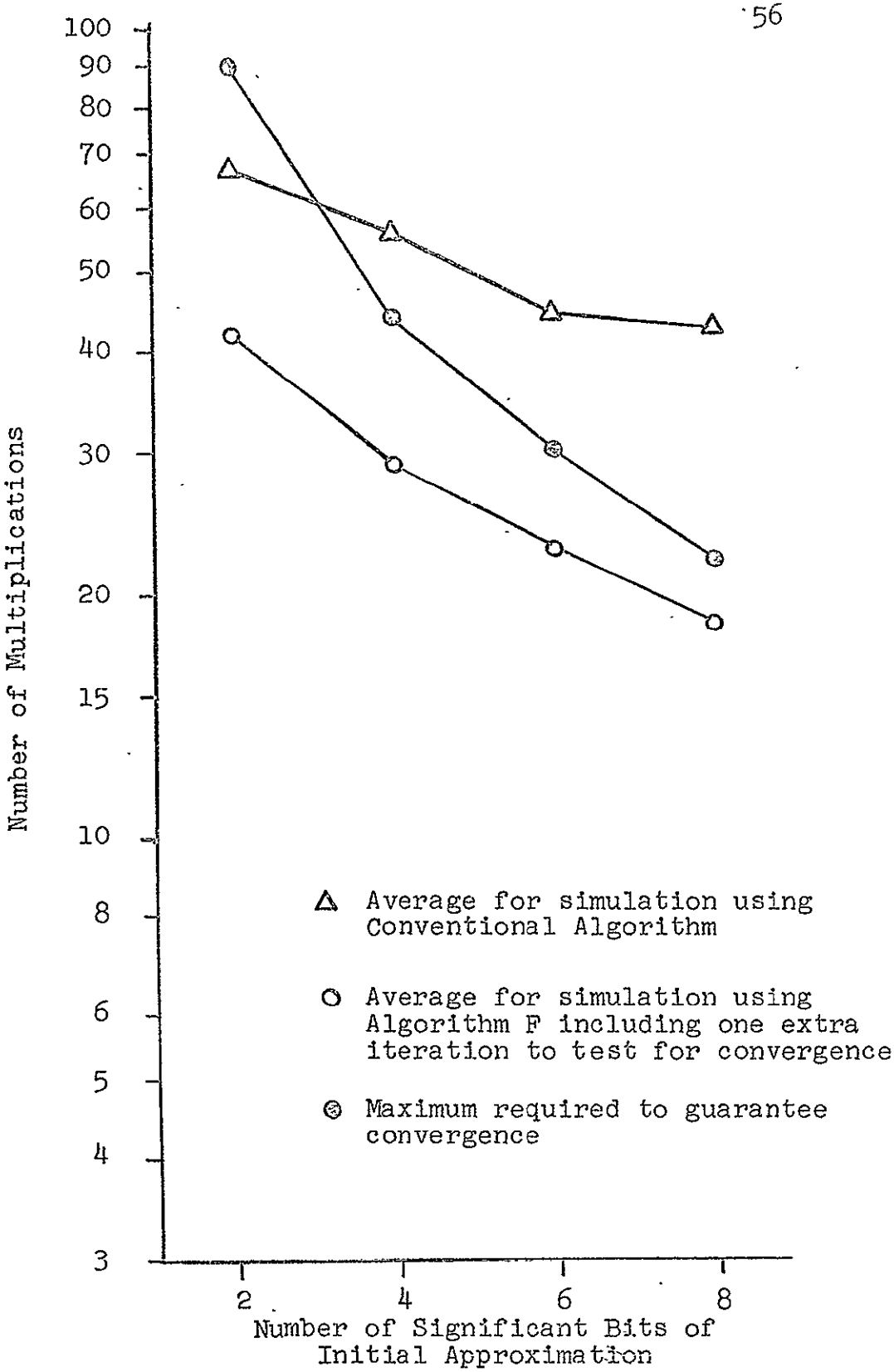


Fig. 6-6 Number of Multiplications for a 92-bit Number vs. Accuracy of Initial Approximation.

CHAPTER 7

COMPARISONS OF ALGORITHM R AND ALGORITHM F

7.1 General Comparisons

Both algorithms R and F provide distinct advantages over conventional methods. Algorithm R is capable of generating the square root with fewer multiplications than a standard method and requires no large look-up table. Algorithm F utilizes existing special hardware similar to that in the IBM 360/91 to more than double the speed of the square root calculation.

On a standard machine a software square root function is a complicated procedure. An initial approximation must be made, calling up constants, an iteration must be made and either a check made for convergence or a counter kept to determine completion. Considering the inherent speed advantages of a hardware implementation, it is felt that either algorithm can significantly decrease computation time for square root without a great addition of hardware.

The two algorithms are hard to compare, because they are designed for different purposes. Algorithm F is intended as an addition to a computer with a multiply/divide unit of the type employed in the 360/91. It

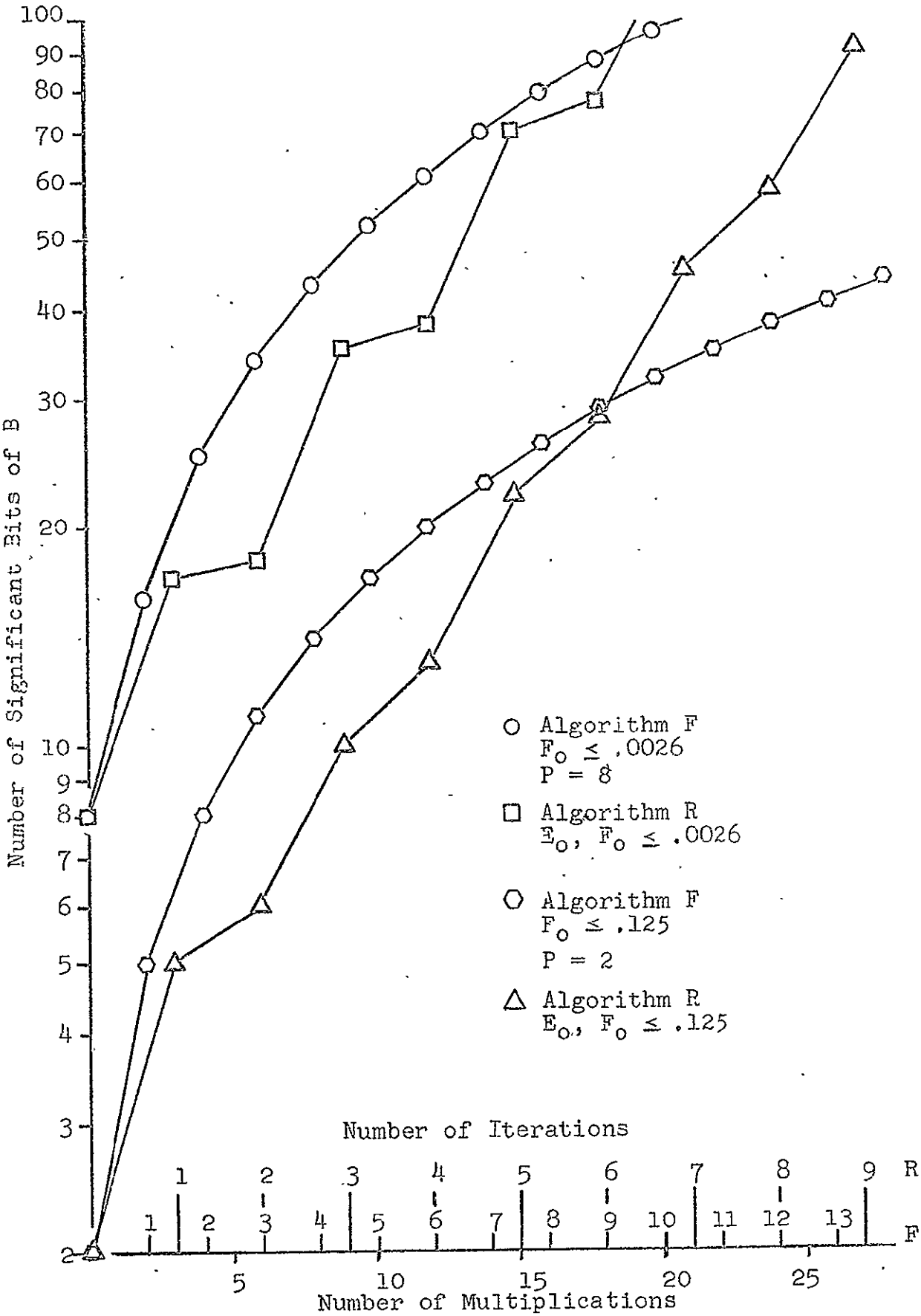


Fig. 7-1 Calculated Minimum number of Significant Bits of B vs. Number of Iterations and Multiplications for Algorithms F and R.

requires little additional hardware. Algorithm R assumes only a fast multiplier and creates a division scheme of its own. As a result, it does require considerable additional hardware.

Some comparisons can be made between the two algorithms. Fig. 7-1 shows convergence rates for the two algorithms. Algorithm F has a steeper slope initially and is much smoother, but due to the fact that it only increases P bits per iteration, its slope declines. On the other hand, although the slope of R is erratic, it nevertheless is fairly constant over several iterations, increasing the number of significant bits by about 55% per iteration. It may be possible to take advantage of its erratic behavior to produce a considerably higher rate, since the values plotted are worst case.

7.2 Comparisons to Conventional Algorithms

An attempt has been made to compare algorithms F and R and also to compare them to the conventional algorithm. The basis of comparison was the minimum number of significant bits generated after each iteration for both algorithms. For purposes of comparison it was assumed that (1) maximum errors for B_o and R_o of algorithm R were the same as maximum errors for B_o of algorithm F

and the conventional algorithm and (2) Algorithm F utilized a reciprocal generator generating the same number of bits as the initial approximation.

The conventional algorithm is not compared directly to the R and F on Fig. 7-1 because of the fact that it requires two convergences--one for the hardware division convergence and one for the software Newton-Raphson square root convergence. However, calculations were made to determine the maximum number of iterations required for convergence, assuming the number of significant bits doubles with each iteration. They are shown in Tables 7-1 and 7-2. In Tables 7-3 and 7-4 they are converted to required number of multiplications--a fairer comparison since R requires 3 multiplications while the conventional altorithm and F require but 2.

7.3 Variance

As mentioned earlier, on the simulation the variance in the number of iterations was calculated. All three algorithms had a low variance with variations seeming more dependent on the particular machine size and initial approximation than anything else. In general, however, surprisingly, R tended to have the lowest variance, particularly on the larger machines. For example, on a 92-bit

TABLE 7-1

MAXIMUM NUMBER OF ITERATIONS REQUIRED TO GENERATE SQUARE ROOT WITH AN 8-BIT INITIAL APPROXIMATION

	Machine Size				
	20	32	48	64	92
Conventional Algorithm	4	4	9	9	16
Algorithm F	2	3	5	7	10
Algorithm R	3	3	5	5	7

TABLE 7-2

MAXIMUM NUMBER OF ITERATIONS REQUIRED TO GENERATE SQUARE ROOT WITH A 2-BIT INITIAL APPROXIMATION

	Machine Size				
	20	32	48	64	92
Conventional Algorithm	16	16	25	25	36
Algorithm F	8	10	16	21	30
Algorithm R	5	7	8	9	9

TABLE 7-3

MAXIMUM NUMBER OF MULTIPLICATIONS REQUIRED TO GENERATE SQUARE ROOT WITH AN 8-BIT INITIAL APPROXIMATION

	Machine Size				
	20	32	48	64	92
Conventional Algorithm	8	8	18	18	32
Algorithm F	4	6	10	14	20
Algorithm R	9	9	15	15	21

TABLE 7-4

MAXIMUM NUMBER OF MULTIPLICATIONS REQUIRED TO GENERATE SQUARE ROOT WITH A 2-BIT INITIAL APPROXIMATION

	Machine Size				
	20	32	48	64	92
Conventional Algorithm	32	32	50	50	72
Algorithm F	16	20	32	42	60
Algorithm R	15	21	24	27	27

machine with 2-bits initial approximation R had a variance of only 0.07 compared to .20 for the conventional algorithm and .22 for F.

The low variances are related to the fact that the average number of iterations was very close to the actual values for both R and F. This indicates that probably a counter will be a more effective method for terminating iteration than a test, particularly for algorithm R which would be very hard to test.

7.4 Round-off Error

None of the iterative methods studied were able to guarantee the last bit. If iteration is continued after convergence, the last bit will frequently alternate and, in the case of algorithm R, the last two bits alternated on occasion. This was attributed to round-off errors caused by the truncation due to multiplication. This error was measured in the simulations by squaring B_k after it had converged and comparing it to N. It was found that both algorithms had smaller errors, on the average than the conventional algorithm, with F slightly better. However, R had a significantly larger variance than the others and its greatest error was generally as large as or larger than the conventional algorithm. This was expected, from observations described earlier.

Although it is difficult to compare any of these square root techniques to direct methods mentioned in Chapter 2, some observations can be made. (1) Direct methods are linear, generating a fixed number of significant bits per unit of time. While Algorithm F is also linear, Algorithm R is nearly second order--it increases significant bits at a higher rate with each iteration. (2) Direct methods are clearly limited in their rate--even complicated techniques used to generate two bits at a time require considerable decoding time On the other hand, algorithm F is not so limited. It can be made to converge at an arbitrary rate by increasing the accuracy of the reciprocal generator and the initial approximation. The breakeven point is when F produces as many significant bits in one iteration as the direct method can in the same time--the time for two multiplications. For example, if the reciprocal generator produces eight significant bits and a multiplication time is 4 cycles, a direct method would have to produce one bit per cycle to be competitive.

CHAPTER 8

IMPLEMENTATIONS OF ALGORITHMS F AND R

Although all of the details of an implementation of algorithms F and R have not been worked out, a possible configuration has been designed for each algorithm.

8.1 Algorithm R

Since algorithm R incorporates its own division algorithm the organization was designed such that both division and square root could be performed rapidly. The multiplier is given a special characteristic not normally found on a multiplier, but not believed difficult to implement. It was given the ability to add in a fixed point number to the product if desired, i.e., it could generate (X * Y) + Z. This would be a valuable ability for other operations as well. It was assumed that the multiplier has three registers, X, Y, and Z. The multiplicand is entered in X while the multiplier is in Y. The product is left in Y which is actually a double length register, although only the most significant half is necessary for square root and division. The Z register contains a number which, upon command, is added to the product during multiplication. This result may be

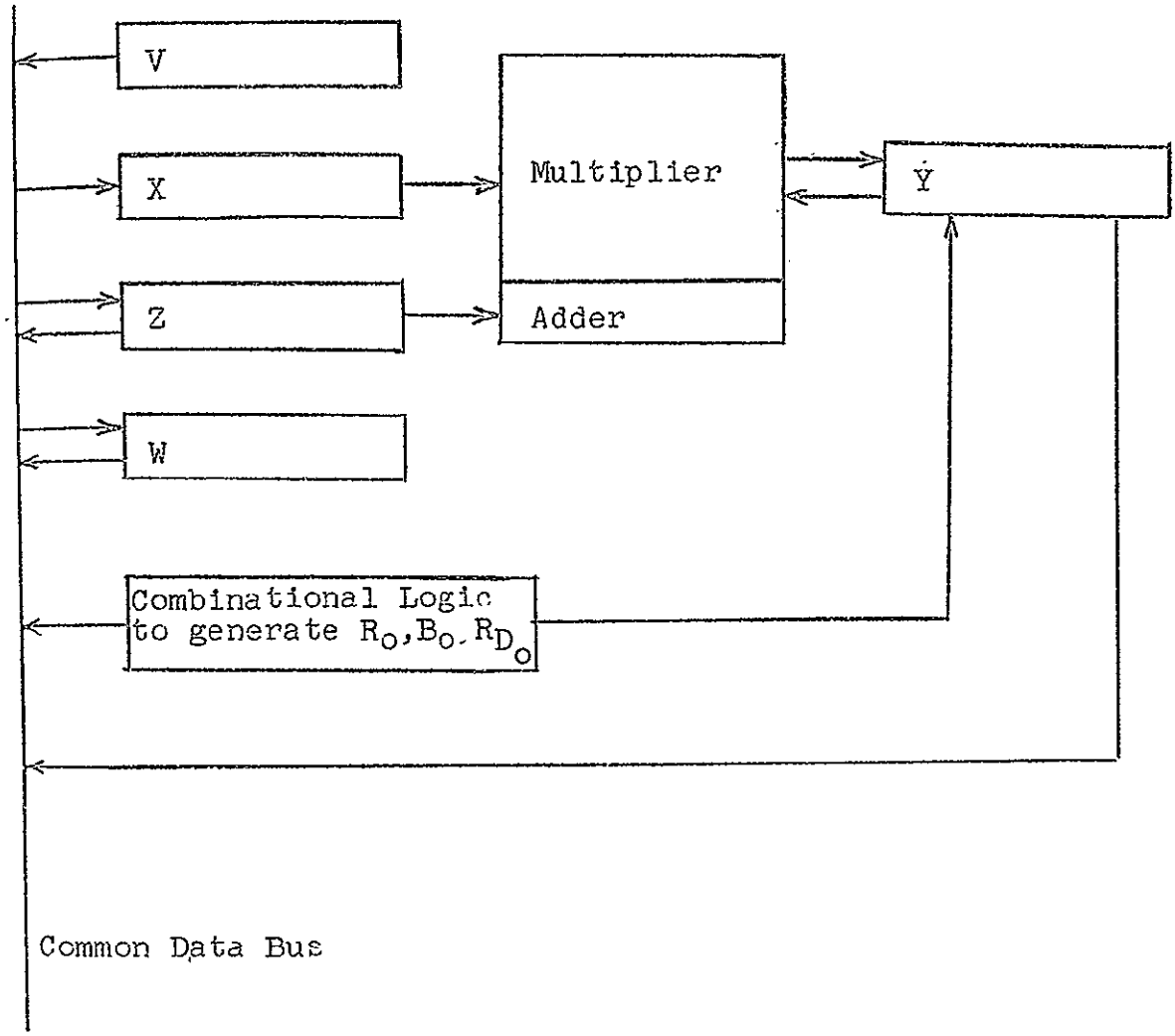


Fig. 8-1 Data Flow for Algorithm R.

delivered shifted right one place, i.e., $\frac{1}{2}[(X) \cdot (Y) + (Z)]$. With this scheme, as shown in Fig. 8-1, both multiplication and division could be expedited very rapidly.

For square root, the operand is initially found in register W. It is assumed that at this point the number has been normalized so that the exponent is ever and the mantissa is greater than or equal to 1/4 and less than 1. A test will have been performed to check for a negative operand, in which case an error diagnostic would be transmitted. In addition, if a zero were detected, the result register will be set to zero and the instruction will be completed.

Register W could be used to generate the addresses in memory for the approximation B_o and R_o. However, since algorithm R has been shown to be not highly dependent on the initial approximation, it would probably be much faster to generate a small initial approximation for B_o and R_o using combinational logic. This approach has been assumed for the implementation.

The sequencing for square root would be as follows:

1) From contents of register W, generate B_o in Y and R_o in X.
2) $(Y) \rightarrow Z$, $\overline{(X) * (Y)} \rightarrow Y$. The value B_o is stored for later use in Z. The 2's complement of R_oB_o is stored in Y.

3) (X) * (Y) $\rightarrow$ Y, (W) $\rightarrow$ X. R_k is being multiplied by the term $2-B_kR_k$ to generate R_{k+1}. Meanwhile the operand N is being moved to X for the next operation.

4) (X) * (Y) + (Z) $\xrightarrow{RSH1}$ Y, (Y) $\rightarrow$ X. The term $B_{k+1} = \frac{1}{2}(B_k + N * R_{k+1})$ is being formed in Y while R_{k+1} is being transferred to X for the next cycle. The contents of Y transferred to X are the contents before the multiplication.

5) Steps 2, 3 and 4 are repeated a fixed number of times until the desired accuracy of B_o is guaranteed.

6) The result B is in register Y while its reciprocal R is in register X.

8.2 Algorithm R--Division

For division little additional hardware is required. A look-up table in combinational logic is assumed for the initial approximation to the reciprocal R_{Do}. The dividend is assumed to be initially in register Z while the divisor is in register W.

1) From contents of register W (divisor) generate R_{Do} in X, and transfer the partial contents of W (same number of significant bits as R_{Do}) to Y.

2) $\overline{(X) * (Y)} \rightarrow Y$. The two's complement term $2 - DR_{D_k}$ is being generated.

3) $(X) * (Y) \rightarrow Y$, $(W) \rightarrow X$. $R_{D_{k+1}}$ is generated while D is being transferred to X for the next iteration.

4) $(X) * (Y) \rightarrow Y$, $(Y) \rightarrow X$. The two's complement term $2 - DR_{D_{k+1}}$ is generated and $R_{D_{k+1}}$ is gated into X for generation of $R_{D_{k+2}}$. The contents of Y transferred to X is the contents before the multiplication.

5) Repeat 3 and 4 until required accuracy for $R_{D_{k+1}} = \frac{1}{D}$ will be obtained on following iteration.

6) $(X) * (Y) \rightarrow Y$, $(V) \rightarrow X$. Generate $\frac{1}{D} = R_{D_{k+1}}$ while the dividend is being transferred for multiplication.

7) $(X) * (Y) \rightarrow Y$, $(Y) \rightarrow X$. The quotient is placed in register Y while its reciprocal is left in X. Again (Y) means before the multiplication.

Steps 6 and 7 are identical to 3 and 4 except that V is gated to X instead of W on step 6 and the product, rather than the two's complement of the product is entered in Y on step 7.

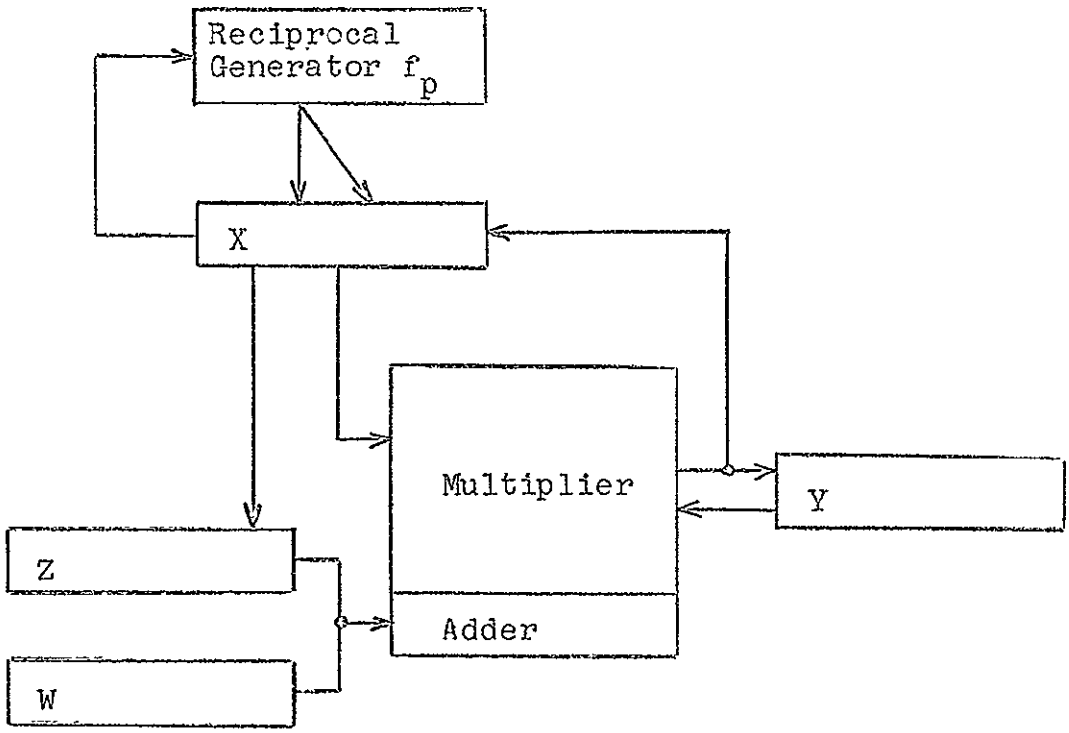


Fig. 8-2 Data Flow for Algorithm R.

8.3 Algorithm F

A reciprocal generator of the type used in the IBM 360/91 was assumed for the implementation of algorithm F shown in Fig. 8-2. As for algorithm R, it was assumed that a sum could be added to a product in the multiplier without a great loss in speed. Although two multiplications are necessary for each iteration, one would be extremely rapid since the multiplier is the P-bit approximation to $\frac{1}{2B_k}$ and contains only P significant bits. This is true even for the final iterations where B_k has a large number of significant bits.

As was assumed in the simulation an approximation was generated for B_o, using constants, α, β, and γ, probably wired, in registers X, Y, and Z respectively. The operand N is in register W and has been checked to see if it is positive. By some means the contents of W is added to X, either through an adder (possibly the adder in the multiplier) or through combinational logic since the sum will have only a few significant bits. This sum is entered in the combinational logic reciprocal generator and its approximate reciprocal is deposited in X. X is multiplied by Y and the sum Z is included. The result, B_o, is stored in Y. The sequence is thus:

1) $f_p[(W) + (X)] \rightarrow X$. Generate $\frac{1}{\gamma+N}$

2) $(X) * (Y) + (Z) \rightarrow X, Y$. Generate $B_o = \alpha + \beta - \frac{1}{\gamma+N}$

3) $f_p(X) \xrightarrow{RSH1} X$, $(X) \rightarrow Z$, $(X) * (Y) + W \rightarrow Y$

Generate $N - B_k^2$, prepare $f_p = \frac{1}{2B_k}$ term in X, by generating reciprocal and gating the output on place right.

4) $(X) * (Y) + (Z) \rightarrow X, Y$. Generate $B_{k+1} = B_k + f_p (N-B_k^2)$.

5) Repeat 3 and 4 until required accuracy is guaranteed. The result, B, is in register Y.

For both algorithms it is possible to speed up the multiplications considerably, during the early iterations because of the small number of significant bits. In addition, for algorithm F, as mentioned earlier, only one multiplication per iteration requires a full multiplication and this would only be required on the final iterations.

For both algorithms some method is required to determine when convergence has been reached. For algorithm F, an easy test is to consider the magnitude of the product $f_p \cdot (N-B_k^2)$. No such simple test exists for R. In addition, for F it was shown that in many cases, the average number of iterations required was so close to the maximum necessary that it is actually faster to do a

fixed number of iterations than to test for convergence. Thus for all cases with R and most cases with F the most efficient way to stop the algorithm is to iterate a predetermined number of times. This would require a small counter.

CHAPTER 9

SUMMARY

9.1 Comparisons

Two algorithms have been proposed which could significantly reduce execution time for the squre root operation. By their nature they are not easy to compare either to each other or to a more conventional method. Algorithm R converges very rapidly, though not quite at a second order rate. The algorithm requires a unique organization but includes a high speed division algorithm. Algorithm F utilizes a special reciprocal generator and converges at a linear rate. It may be forced to converge at any rate desired by brute force, i.e., its rate is dependent on the accuracy of the reciprocal generator and the initial approximation. Both algorithms are believed to have some profitable applications, speeding up square root execution without a great expense of hardware.

The comparison of algorithms F and R with a conventional algorithm was based on the number of multiplications required. In some respects, this may be an unfair comparison, because division execution time is considerably

less than the product of the number of multiplications and the multiplication execution time. For example, in the IBM 360/91, despite the fact that a multiplication requires 6 clock cycles and 9 multiplications are required for division, only 18 clock cycles are necessary, i.e., two per multiplication. This is a result of the fact that all but the last multiplication require only part of the multiplier because they involve fewer significant bits. It is believed that the same techniques can be used with algorithms R and F to speed up square root time, and for algorithm R to speed up division.

Both algorithms were shown by simulation to be faster than conventional iterative techniques, particularly on large machines. In particular, algorithm R was shown to have its greatest advantage for a bad initial approximation, since its convergence rate was not greatly affected. These results were also shown from calculations of the maximum number of iterations required to guarantee an arbitrary accuracy.

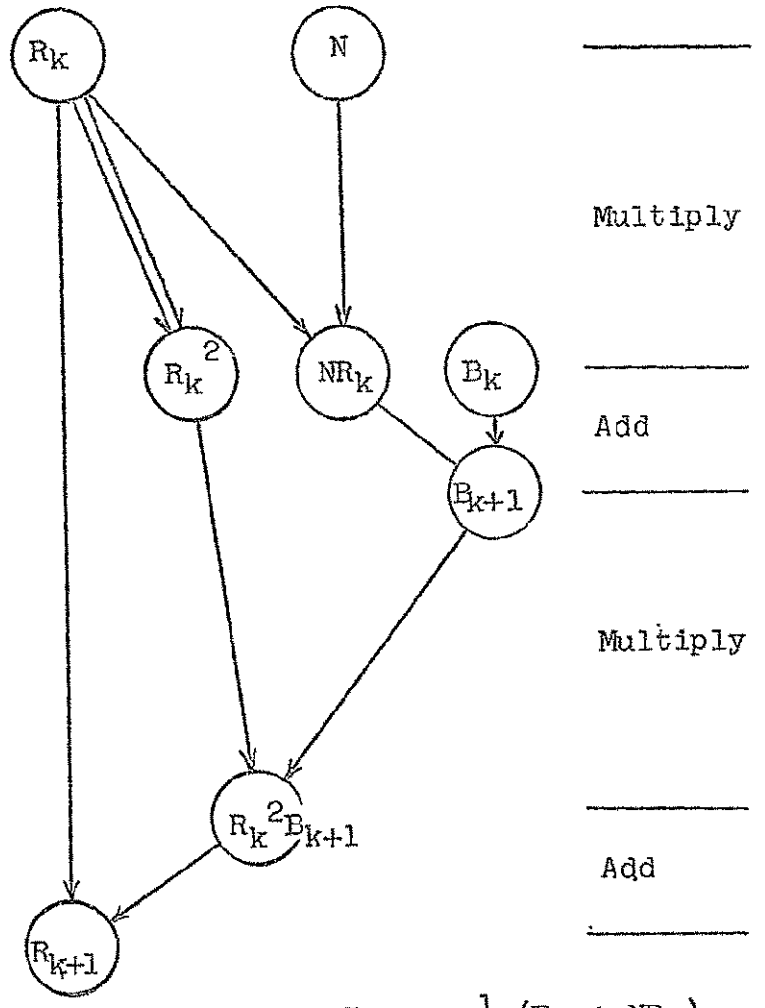
9.2 Possible Modifications

In addition to the implementation suggested, a number of other possibilities exist. If more than one multiplier were available, an additional speedup would

be possible for algorithm R. However, by the nature of the algorithm, two multiplications would still have to be performed alternately as shown in Fig. 9-1 unless a multiplication of three factors were available.

With some modifications, algorithm F might be made to converge at a nearly second order rate with a second multiplier available. While the B_k^2-N term was being generated, the reciprocal approximation $f = \frac{1}{2B_k}$ could be used to generate a better approximation by use of the Newton-Raphson reciprocal iteration, $f = f\ (2-2fB_k)$, or some modification. This approximation to $\frac{1}{2B_{k+1}}$ will clearly be better than the approximation from the generator which will not change from $\frac{1}{2B_k}$

On a machine for which multiplication is very fast compared to division, R might be micro-programmed or even implemented completely in software. It is possible, also, that a machine such as the 360/91, containing a table look-up reciprocal generator, might be micro-programmed to do square root by algorithm F. For example, with an eight-bit initial approximation, algorithm F would require only 6 iterations to converge. The use of the same approximation and the Newton-Raphson iteration would require 4 divisions. Algorithm F requires an extra



$$B_{k+1} = \frac{1}{2}\,(B_k + NR_k)$$

$$R_{k+1} = R_k\,(2 - B_{k+1}R_k)$$

$$= 2R_k - R_k^2 B_{k+1}$$

Fig. 9-1 Sequence Diagram for Algorithm R Using Two Multipliers.

addition but no shift for the factor 1/2 in the Newton-Raphson iteration. Thus, for a six-cycle multiply and an 18-cycle divide, each would take 72 cycles plus the time required for other manipulations. If any speed-up in multiplication could be achieved, algorithm F would be faster. For example, if the multiplication involving f_p were reduced to two cycles, the multiplication time would be only 48 cycles or 2/3 as much.

9.3 Topics for Further Study

The use of multiplication as a recursive operator for square root extraction demonstrates that complex functions can be generated rapidly by these techniques. Many algorithms of this type could be developed for other functions. Algorithms F and R might be generalized so that it is possible to generate an arbitrary root by this method. In addition, this approach may be valuable in generating other functions which can be found by iterative techniques.

Another topic for further study would be the use of more than one multiplier. It was shown that both algorithms can be improved by the use of multiple units, though neither was developed with this in mind. Quite possibly a more efficient square root technique could be implemented if additional multiplication units were available.

REFERENCES

1. Anderson, S. F., J. G. Earle, R. E. Goldschmidt, D. M. Powers, "The IBM System/360 Model 91: Floating Point Execution Unit," IBM Journal, January, 1967.

2. Barnes, George H., Richard M. Brown, Majo Kato, David J. Kuck, Daniel L. Slotnick, Richard A. Stokes, "The ILLIAC IV Computer," IEEE Transactions on Computers, Vol. C-17, No. 8, August, 1968.

3. Control Data Corporation, 6600 Computer System/Library Function Manual Publication #601 14500, Rev. A., 1965.

4. Flores, Ivan; The Logic of Computer Arithmetic, Englewood Cliffs, New Jersey, Prentice-Hall, Inc., 1963.

5. Maehly, Hans J., "Approximations for the Control Data 1604," Control Data Corporation, March, 1960.

6. Scientific Data Systems 920/930 Programmed Operators Manual, Publication #SDS 900020C, January, 1965.

TWO SQUARE ROOT ALGORITHMS UTILIZING MULTIPLICATION AS THE ITERATIVE OPERATOR

University of Texas
NGR-44-012-144

APPROVED:

_____

_____