



Optimal Size Integer Division Circuits *

John H. Reif

Stephen R. Tate

Department of Computer Science

Duke University

Durham, NC 27706

Abstract

Division is a fundamental problem for arithmetic and algebraic computation. This paper describes boolean circuits (of bounded fan-in) for integer division (finding reciprocals) that have size $O(M(n))$ and depth $O(\log n \log \log n)$, where $M(n)$ is the size complexity of $O(\log n)$ depth integer multiplication circuits. Currently, $M(n)$ is known to be $O(n \log n \log \log n)$, but any improvement in this bound that preserves circuit depth will be reflected by a similar improvement in the size complexity of our division algorithm. Previously, no one has been able to derive a division circuit with size $O(n \log^c n)$ for any c , and simultaneous depth less than $\Omega(\log^2 n)$. Our circuits are logspace uniform; that is, they can be constructed by a deterministic Turing machine in space $O(\log n)$.

Our results match the best known depth bounds for logspace uniform circuits, and are optimal in size.

The general method of high order iterative formulas is of independent interest as a way of efficiently using parallel processors to solve algebraic problems. In particular, our algorithm implies that any rational function can be evaluated in these complexity bounds.

As an introduction to high order iterative methods we also present a circuit for finding polynomial reciprocals (where the coefficients come from an arbitrary ring, and ring operations are unit cost in the circuit) in size $O(PM(n))$ and depth $O(\log n \log \log n)$, where $PM(n)$ is the size complexity of optimal depth polynomial multiplication.

1 Introduction

In arithmetic and algebraic computation, the basic operations are addition, subtraction, multiplication, and division. It is a fundamental problem to find efficient algorithms for division, as it seems to be the most difficult of these basic operations. Problems are studied with both sequential models (Turing machines or bit-operation RAMs) and parallel models (circuits and bit-operation PRAMs); the model that we use in this paper is the circuit. A circuit is an acyclic directed graph with a set of nodes designated as input nodes (with zero fan-in), a set of nodes designated as output nodes (with zero fan-out), and a function basis with the elements labeling all non-input nodes. The value at any node is computed by applying the function labeling that node to the values of its predecessors, which are found in the same way – this goes on recursively until the input nodes are reached. Assigning a vector of values to the input nodes and computing the value of each output node, we see that a circuit computes a function over vectors in the value domain. All circuits discussed in this paper have the additional restriction that every node must have fan-in bounded by some constant (without loss of generality, we can assume that every node has no more than 2 predecessors). The *size* of a circuit is the number of nodes in the circuit, and the *depth* of the circuit is the length of the longest path from an input node to an output node. The circuits used in most of this paper have function basis made up of the boolean functions AND, OR, and NOT; these are called bounded fan-in boolean circuits and are the standard model for arithmetic computation. In section 3 (dealing with polynomial reciprocals) we use a circuit model with operations in an arbitrary ring as the basis.

Optimal algorithms have been known for quite some time for addition and subtraction, and good algorithms exist for multiplication. If we let $SM(n)$ be the sequential time complexity of multiplication and $M(n)$ be the size complexity of $O(\log n)$ depth multiplication using the circuit model, then the best known results are due to Schönhage and Strassen [11] who give an algorithm based on discrete Fourier transforms with $M(n) = O(SM(n)) = O(n \log n \log \log n)$.

The problem of integer division was examined by

* This research was supported by National Science Foundation Grant CCR-8696134, by grants from the Office of Naval Research under contracts ONR-N00014-87-K-0310 and ONR-N00014-88-K-0458, by DARPA contract DAAL03-88-K-0195, and by Air Force contract AFSOR-87-0386.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

S. Cook in his Ph.D. thesis [5], and it was shown by using second order Newton approximations that the sequential time complexity of taking reciprocals is asymptotically the same as that of multiplication. Unfortunately, this method does not carry over to the circuit model – for size $O(M(n))$ division circuits, we require depth $\Omega(\log^2 n)$ from a direct translation of Cook's method of Newton iteration. In addition, no one has been able to derive a new method for integer division with size $O(M(n))$ and depth less than $\Omega(\log^2 n)$ until now.

It was a long-standing open question to match the optimal depth bounds obtained for addition, subtraction, and multiplication with a division circuit of polynomial size. Until 1983, no one had presented a circuit for finding reciprocals with polynomial size and depth better than $\Omega(\log^2 n)$, then J. Reif presented a logspace uniform circuit based on wrapped convolutions with depth $O(\log n (\log \log n)^2)$ and polynomial size [8]. A year later Beame, Cook, and Hoover presented a polynomial time uniform circuit based on Chinese Remaindering with depth $O(\log n)$ [3]. A revised paper by Reif reduced the depth bounds on the logspace uniform circuit to $O(\log n \log \log n)$ [9]. For giving deterministic space bounds, logspace uniform circuits are vital as explained by Borodin [4]; in addition, the polynomial time uniform circuits that have been given use polynomial size tables of precomputed values, which a purist might find objectionable.

The size bounds for the above circuits are at least quadratic, and further work has been done to decrease the size bounds while keeping the depth the same. Shankar and Ramachandran [12] make a significant step in this direction by use discrete Fourier transforms to reduce the problems in size. They then apply either Reif's circuit (to give a logspace uniform circuit), or the Beame, Cook, and Hoover circuit (to give a polynomial time uniform circuit). The best depth bounds for each type of circuit are matched, and the size of both circuits is $O(\frac{1}{\epsilon} n^{1+\epsilon})$, for any sufficiently small $\epsilon > 0$. Independent work on a polynomial time uniform circuit by Hastad and Leighton [6] resulted in an efficient circuit for Chinese Remaindering which gave a division circuit of size $O(n^{1+\epsilon})$ and depth $O(\frac{1}{\epsilon} \log n)$, for $\epsilon > 0$.

Until 1988, no one had given a circuit with depth less than $\Omega(\log^2 n)$, and simultaneous size $O(n \log^c n)$ for any c . A preliminary version of this paper [10] gave logspace uniform circuits that have size $O(M(n))$ and depth $O(\log n (\log \log n)^2)$. Now we improve these results and present logspace uniform circuits that have size $O(M(n))$ and depth $O(\log n \log \log n)$. We use Newton approximations of high degree to gain as many bits as possible in the early stages, and thus reduce the overall number of stages required. We further note that the size bound of our circuit is asymptotically tight (within a constant factor) with the optimal size bound of multiplication, so further improvements in multiplication

would be mirrored by improvements in integer division. Furthermore, by a classic result given in Aho, Hopcroft, and Ullman [1], multiplication can be done with a constant number of reciprocals, so our circuit has optimal size, while matching the best known depth bounds for logspace uniform circuits. A result of Alt [2] immediately applies to our results to give as a corollary that any rational function can be evaluated in $O(M(n))$ size and $O(\log n \log \log n)$ depth.

We will first show how to compute reciprocals of polynomials in size $O(PM(n))$ and depth $O(\log n \log \log n)$, where $PM(n)$ is the size complexity of $O(\log n)$ depth polynomial multiplication. The polynomial problem provides a good introduction to the high order iterative methods that we use. High order iterative methods date back to Euler; a general discussion of high order iteration formulas can be found in Traub [13]. The method of using high order Newton approximations is of independent interest as a way of efficiently using processors in a parallel system.

2 Algorithm Overview

In Cook's reduction of division to multiplication, he used second order Newton approximations with each successive stage dealing with twice the number of bits as its predecessor. The sequential complexity of a single stage of second order Newton iteration is $O(SM(n))$. Since $SM(n)$ must be at least linear, the geometric progression of approximation lengths makes the sum over all stages no more than $O(SM(n))$. However, the circuit model of multiplication has size $M(n)$ and depth $O(\log n)$, and both size and depth must be summed over all stages. The same effect is noticed with the geometrically decreasing sizes, and the overall size of Cook's division algorithm is $O(M(n))$. Unfortunately, since the depth is only logarithmic, the fact that n is geometrically decreasing is not enough to keep the total depth from increasing to $\Omega(\log^2 n)$ in the summation.

Our key observation was that since the size and depth of the first stages in Cook's algorithm are so small, considerably more work can be done than a simple second order approximation. We divide our algorithm into two parts: part A uses high order Newton approximations, and part B extends this result to n bits using $O(\log \log n)$ second order approximations. We present a formula for calculating the k th order Newton iteration for the reciprocal problem which increases the accuracy of an approximation (in bits) by a factor of k . In the early stages, we can afford to make k large, and do much more work on each stage than simply doubling the number of bits as done by Cook.

We select the value of k for a particular stage by making the size of every stage meet the same bound. The results we get are that the number of approximation stages required drops from $\Omega(\log n)$ to $O(\log \log n)$

for both integer reciprocals and polynomial reciprocals. The required number of iterations is heavily influenced by the size complexity of taking large powers, and for integer powering we present a new, size efficient powering algorithm.

3 Use of High Order Iterations for Polynomial Inverse

If we let $R = \{D, +, \cdot, 0, 1\}$ be an arbitrary ring, then we define a polynomial $p(x)$ of degree $n-1$ in $R[X]$ to

be $p(x) = \sum_{i=0}^{n-1} a_i x^i$. In this section we will often define a

new polynomial of degree $k-1$ by using the coefficients of the k highest degree terms of a higher degree polynomial. We denote the degree $k-1$ polynomial derived in

this way from $p(x)$ by $p_k(x) = \sum_{i=0}^{k-1} a_{n-k+i} x^i$. In prob-

lems dealing with polynomials we use the bounded fan-in circuit model, but allow each node to compute either addition, multiplication, or reciprocation in the ring R in unit size and unit depth. Notice that since reciprocation is allowed, some computations may be undefined.

The polynomial reciprocal problem as defined in Aho, Hopcroft, and Ullman [1] is to calculate a polynomial $q(x)$ from a $(n-1)$ st degree polynomial $p(x) \in R[X]$ such that

$$q(x) = \text{RECIPROCAL}(p(x)) = \left\lfloor \frac{x^{2n-2}}{p(x)} \right\rfloor.$$

As previously mentioned, high order iterative methods take an estimate of length d , and produce a new estimate of length kd . In the case of polynomials, we use $\text{RECIPROCAL}(p_d(x))$ as the length d "estimate" - notice that $\text{RECIPROCAL}(p(x)) = \text{RECIPROCAL}(p_n(x))$. To produce the estimate of length kd , first calculate the intermediate polynomial

$$r(x) = s(x) \sum_{j=0}^{k-1} \left[x^{(k+1)d-2} \right]^{k-j-1} \left[x^{(k+1)d-2} - p_k(x)s(x) \right]^j \quad (1)$$

where $s(x) = \text{RECIPROCAL}(p_d(x))$. Now

$$\text{RECIPROCAL}(p_k(x)) = \left\lfloor \frac{r(x)}{x^{(k-1)(kd-2)}} \right\rfloor \quad (2)$$

Lemma 3.1 Given $s(x) = \text{RECIPROCAL}(p_d(x))$, equation (2) produces $\text{RECIPROCAL}(p_k(x))$, and can be evaluated in $O(k^3 d \log(kd))$ size and $O(\log(kd))$ depth.

PROOF: First we prove the correctness of the iteration formula. We state the lemma in a different (but equivalent) way; that is, that equation (2) produces a polynomial $q(x)$ such that $q(x)p_k(x) = x^{2kd-2} + t(x)$

where $t(x)$ is some polynomial of degree less than $kd-1$. The polynomial $s(x)$ satisfies $p_d(x)s(x) = x^{2d-2} + t_1(x)$, where $\text{degree}[t_1(x)] < d-1$.

Since $p_k(x) = p_k(x)x^{k(d-1)} + p'(x)$ (where $\text{degree}[p'(x)] \leq k(d-1)-1$), multiplying by $s(x)$ gives $p_k(x)s(x) = x^{(k+1)d-2} + x^{(k-1)d}t_1(x) + s(x)p'(x)$. For simplicity of notation we let $f(x) = x^{(k+1)d-2}$ and $g(x) = -(x^{(k-1)d}t_1(x) + s(x)p'(x))$, so $p_k(x)s(x) = f(x) - g(x)$. Using this notation the iteration formula gives

$$\begin{aligned} p_k(x)r(x) &= f(x) \sum_{j=0}^{k-1} [f(x)]^{k-j-1} [g(x)]^j - \\ &\quad g(x) \sum_{j=0}^{k-1} [f(x)]^{k-j-1} [g(x)]^j \\ &= \sum_{j=0}^{k-1} [f(x)]^{k-j} [g(x)]^j - \sum_{j=1}^k [f(x)]^{k-j} [g(x)]^j \\ &= [f(x)]^k - [g(x)]^k \\ &= x^{(k+1)kd-2k} - \left[-(x^{(k-1)d}t_1(x) + s(x)p'(x)) \right]^k \end{aligned}$$

Now doing the division by $x^{(k-1)(kd-2)}$ (which is actually just a shift of coefficients) and discarding the remainder, we get

$$\begin{aligned} p_k(x)q(x) &= x^{2kd-2} - \left\lfloor \frac{[-(x^{(k-1)d}t_1(x) + s(x)p'(x))]^k}{x^{(k-1)(kd-2)}} \right\rfloor \\ &= x^{2kd-2} + t(x) \end{aligned}$$

Examining the degrees of the components of $t(x)$ we see that the numerator is just $g(x)$ which, on closer examination, satisfies $\text{degree}[g(x)] \leq kd-2$. After the powering of $g(x)$ we see that $\text{degree}[g(x)]^k \leq k^2 d - 2k$. Now we look at the division to see that $\text{degree}[t(x)] \leq kd-2$, and the correctness of our formula is proved.

To determine the complexity of the iteration formula, first note that a polynomial of degree $kd-1$ can be raised to the k th power in size $O(k^2 d \log(kd))$ and depth $O(\log(kd))$ by using discrete Fourier transforms (see, for example [9]). There are k different powers to take and add up, so this cost dominates the entire calculation. The total size is $O(k^3 d \log(kd))$, and the total depth is $O(\log(kd))$. \square

We repeatedly apply the iteration formula of equation (2) to get the complete polynomial reciprocal algorithm. The details are described in the proof of the following theorem.

Theorem 3.1 The reciprocal of an $(n-1)$ st degree polynomial $p(x)$ as defined above can be computed in $O(PM(n))$ size and $O(\log n \log \log n)$ depth, where $PM(n)$ is the size complexity of $O(\log n)$ depth polynomial multiplication.

PROOF: Without loss of generality, we can assume that n is a power of two as explained in Aho, Hopcroft, and

Ullman [1]; in particular, we let $n = 2^m$ for some integer m . If we let $f(i) = \lceil m(1 - (\frac{2}{3})^{i-1}) \rceil$, then we can define a schedule of d values as $d_i = 2^{f(i)}$; notice that $d_1 = 1$. Letting $p(x) = a_n x^{n-1} + p'(x)$, then a_n^{-1} is the reciprocal of the degree 0 polynomial that serves as the base of our algorithm.

The order of the iteration formula that we use at stage i is $2^{f(i)-f(i-1)}$, and it is easy to show that $f(i) - f(i-1) \leq \frac{m}{3} (\frac{2}{3})^{i-1} + 1$. Substituting actual values for d_i and k_i in the complexity bounds, stage i takes size $O(n \log n)$ and depth $O(\log n)$. The number of iterations is $O(\log \log n)$ (this is easy to see – just solve $m(1 - (\frac{2}{3})^{i-1}) = m - 1$), so the total size of all stages of this algorithm is $O(n \log n \log \log n)$, and the total depth is $O(\log n \log \log n)$.

The algorithm that was just described is not quite what we use though. If we take the first $\frac{n}{\log^2 n}$ coefficients of $p(x)$ and find the reciprocal of the polynomial defined by these coefficients, then letting $n' = \frac{n}{\log^2 n}$ we can do this by the previously mentioned algorithm in size $O(n' \log n' \log \log n') = O(n)$ and depth $O(\log n \log \log n)$. This is part A of our polynomial reciprocal algorithm.

Part B is a series of second order iterations, and the required number of stages is $O(\log \log n)$. Part B is easily seen to have size $O(PM(n))$ and depth $O(\log n \log \log n)$. The total complexity of our polynomial reciprocation algorithm is the sum of parts A and B, so the total size is $O(PM(n))$, and the total depth is $O(\log n \log \log n)$. \square

4 High Order Iteration For Integer Division

The following definition is useful when describing the amount of error present in an approximation.

Definition 4.1 An approximation \tilde{x} to a value x is said to be accurate to c bits if $|x - \tilde{x}| \leq 2^{-c}$.

Note that this definition is the intuitive definition of “accurate to c bits in the fractional part.” The reciprocal problem is that given a value x , we need to find the value $y = 1/x$ to within a certain error bound. We will scale the input so that $1/2 < x \leq 1$, which has no effect on the problem – the result will simply be scaled back at the end. The complexity is also not affected since the scaling can be done by powers of 2 (which can be done by bit shifting). If the scaled value of x is accurate to n bits, then we want y accurate to n bits.

Newton iteration is a general method of refining a guess to the exact answer of a problem. The second order Newton iteration formula for finding reciprocals has been known and used for quite some time (see, for example, [5]). What we use in this paper are Newton

iterations of higher degree. In general, a k th order Newton iteration for the reciprocal problem is given by

$$y_{i+1} = y_i \sum_{j=0}^{k-1} (1 - x y_i)^j,$$

where the values y_i are the approximations to y .

In the following error analysis we let $\epsilon_{y,i}$ be the difference between y and the approximation y_i at step i , so $y_i = y - \epsilon_{y,i}$.

Theorem 4.1 If the error at step i is $\epsilon_{y,i}$, then after applying a k th order Newton iteration, the error at step $i+1$ satisfies the inequality $|\epsilon_{y,i+1}| \leq |\epsilon_{y,i}|^k$.

PROOF: Rewriting y_i as $y - \epsilon_{y,i}$, the Newton sum can be rewritten:

$$y_{i+1} = (y - \epsilon_{y,i}) \sum_{j=0}^{k-1} (1 - x(y - \epsilon_{y,i}))^j = (y - \epsilon_{y,i}) \sum_{j=0}^{k-1} (x \epsilon_{y,i})^j$$

since $xy = 1$. Further simplifications give:

$$\begin{aligned} y_{i+1} &= y \sum_{j=0}^{k-1} (x \epsilon_{y,i})^j - \epsilon_{y,i} \sum_{j=0}^{k-1} (x \epsilon_{y,i})^j \\ &= y + \sum_{j=0}^{k-2} x^j \epsilon_{y,i}^{j+1} - \sum_{j=0}^{k-1} x^j \epsilon_{y,i}^{j+1} \\ &= y - x^{k-1} \epsilon_{y,i}^k \end{aligned}$$

Since we have $x \leq 1$, this obviously implies that $|\epsilon_{y,i+1}| \leq |\epsilon_{y,i}|^k$. \square

In our algorithm we will use only iterations of even degree because of the nice ordering properties of even degree approximations. The following obvious corollary shows the relationship between y_{i+1} and y .

Corollary 4.1 If k is even, then after applying a k th degree Newton iteration at step i , $y_{i+1} < y$.

In the discussion above, we assumed that we were performing calculations with all the bits of x (i.e., x has infinite precision). A natural question to ask is how many bits of x we really need to consider to achieve the desired error bound of $|\epsilon_{y,i+1}| \leq |\epsilon_{y,i}|^k$. We answer this question in the remainder of this section.

First, let's introduce some more notation. We will be taking only the most significant bits of x and throwing away the least significant bits. The truncated value is called \tilde{x} , and $\tilde{y} = 1/\tilde{x}$. It is trivial to see that $\tilde{x} \leq x$, so $\tilde{y} \geq y$. We will let $\epsilon_{\tilde{x}} = x - \tilde{x}$, and $\epsilon_{\tilde{y}} = \tilde{y} - y$.

Lemma 4.1 If we know that $|\epsilon_{y,i}| \leq c \leq \frac{1}{4}$ for some value c , and can insure that $\epsilon_{\tilde{y}} \leq c^k$, then performing the k th order iteration (k even) will result in $|\epsilon_{y,i+1}| \leq c^k$.

PROOF: It is important to notice that we are doing the exact Newton iteration for \tilde{y} . We will consider three cases, one for each possible ordering of y , \tilde{y} , and y_i .

Case 1: $y \leq \tilde{y} < y_i$

As we noted in the preceding corollary, after performing the iteration $y_{i+1} \leq \tilde{y}$. Since $\tilde{y} - y \leq c^k$ and $\tilde{y} - y_{i+1} \leq |\epsilon_{y,i}|^k \leq c^k$, it follows that $|y - y_{i+1}| \leq c^k$.

Case 2: $y \leq y_i \leq \tilde{y}$

After the Newton iteration the order must be $y \leq y_{i+1} \leq \tilde{y}$, and since $\tilde{y} - y \leq c^k$, then $|y - y_{i+1}| \leq c^k$.

Case 3: $y_i < y \leq \tilde{y}$

After the Newton iteration either $y \leq y_{i+1} \leq \tilde{y}$ (and $|y - y_{i+1}| \leq c^k$ as in case 2), or $y_{i+1} < y \leq \tilde{y}$. Considering the latter ordering, $\tilde{y} - y_i = \epsilon_{y,i} + \epsilon_{\tilde{y}}$, so $\tilde{y} - y_{i+1} \leq (\epsilon_{y,i} + \epsilon_{\tilde{y}})^k$ and $y - y_{i+1} \leq (\epsilon_{y,i} + \epsilon_{\tilde{y}})^k - \epsilon_{\tilde{y}}$. Furthermore,

$$\begin{aligned} (\epsilon_{y,i} + \epsilon_{\tilde{y}})^k - \epsilon_{\tilde{y}} &= \sum_{j=0}^k \binom{k}{j} \epsilon_{\tilde{y}}^j \epsilon_{y,i}^{k-j} - \epsilon_{\tilde{y}} \\ &= \epsilon_{y,i}^k + \epsilon_{\tilde{y}} \left(\sum_{j=1}^k \binom{k}{j} \epsilon_{\tilde{y}}^{j-1} \epsilon_{y,i}^{k-j} - 1 \right) \end{aligned}$$

Now look at the sum

$$\begin{aligned} \sum_{j=1}^k \binom{k}{j} \epsilon_{y,i}^{k-j} \epsilon_{\tilde{y}}^{j-1} &\leq \sum_{j=1}^k \binom{k}{j} c^{k-j} (c^k)^{j-1} \\ &= \sum_{j=1}^k \binom{k}{j} c^{j(k-1)} \\ &< c^{k-1} \sum_{j=1}^k \binom{k}{j} \\ &\leq 2^{-2(k-1)} (2^k - 1) \\ &= 2^{-k+2} - 2^{-2(k-1)} \\ &\leq 1 \quad \text{for all } k \geq 1 \end{aligned}$$

This implies that $y - y_{i+1} \leq \epsilon_{y,i}^k \leq c^k$, and $|y - y_{i+1}| \leq c^k$. \square

The following theorem sums up the point of the entire section.

Theorem 4.2 If we have y_i accurate to p bits with $p \geq 2$, then applying a k th order Newton iteration (where k is even) using the first $kp + 2$ bits of x results in y_{i+1} accurate to kp bits.

PROOF: y_i is accurate to $p \geq 2$ bits means that $|\epsilon_{y,i}| \leq 2^{-p} \leq \frac{1}{4}$. Let $c = 2^{-p}$ and notice that $\epsilon_{\tilde{x}} \leq 2^{-(kp+2)} = \frac{1}{4} c^k$. Now look at $\epsilon_{\tilde{y}}$.

$$\epsilon_{\tilde{y}} = \frac{1}{\tilde{x}} - \frac{1}{x} = \frac{x - \tilde{x}}{\tilde{x}x} = \frac{x - (x - \epsilon_{\tilde{x}})}{\tilde{x}x} = \frac{\epsilon_{\tilde{x}}}{\tilde{x}x}$$

Since $x \geq \tilde{x} \geq \frac{1}{2}$, we know that $\epsilon_{\tilde{y}} \leq 4\epsilon_{\tilde{x}} \leq c^k$. Now lemma 4.1 directly applies to give $|\epsilon_{y,i+1}| \leq c^k = 2^{-kp}$, and we see that y_{i+1} is accurate to kp bits. \square

5 The Complexity Of Each Step

In this section we derive size and depth bounds for refining a p bit approximation to pk bits. As seen in the previous section, a k th degree Newton iteration (assume k is even from here on) on a p bit approximation yields a new approximation of at least pk bits when the first $pk + 2$ bits of x are used. Therefore we first determine the complexity of a k th order Newton iteration, using pk bits, then see what happens when 2 more bits are used.

To calculate the required approximations, we use a new method of powering by discrete Fourier transforms of the form introduced in Reif [9], and later significantly improved in size by Shankar and Ramachandran [12]. The results are stated in the following theorem. For more information, see appendix A.

Theorem 5.1 We can calculate the k th order Newton iteration on a p bit number (using pk bit calculations) by a logspace uniform circuit of size $O(pk^2 \log p \log \log p)$ and depth $O(\log p + \log k \log \log k)$.

The calculations that follow do not guarantee that k is an integer. In such a case, we perform an order $\lceil k \rceil$ Newton iteration, which will produce an approximation accurate to at least pk bits. Adding one or two to k , if needed to take the ceiling or make it even, obviously does not affect the asymptotic bounds. Similarly, doing calculations with $pk + 2$ bits does not affect the asymptotic bounds. From these facts and theorems 5.1 and 4.2, we get the following corollary.

Corollary 5.1 An approximation accurate to pk bits can be obtained from a p bit approximation by a logspace uniform circuit of size $O(pk^2 \log p \log \log p)$ and depth $O(\log p + \log k \log \log k)$.

6 The Integer Reciprocal Algorithm

In this section, we get to the heart of the reciprocal algorithm. As mentioned in the overview of the algorithm, in part A of our algorithm we choose the highest degree Newton approximation possible, while staying within given size bounds. We let p_i denote the number of bits of accuracy at stage i , and we can define a schedule of accuracies by $p_i = n^{1 - (\frac{1}{2})^i}$; notice that $p_0 = 1$ (only one bit is known initially).

Theorem 6.1 Part A of the reciprocal algorithm calculates the reciprocal of x accurate to $\frac{n}{(\log n)^2}$ bits in $O(n)$ size and $O(\log n \log \log n)$ depth.

PROOF: From the formulas for p_i and p_{i+1} , we can easily solve to see what degree Newton iteration we need

at stage i – call this k_i .

$$k_i = \frac{p_{i+1}}{p_i} = n^{1-(\frac{1}{2})^{i+1}-(1-(\frac{1}{2})^i)} = n(\frac{1}{2})^{i+1}$$

Now we can derive the size complexity of step i to be bounded by

$$cp_i k_i^2 \log p_i \log \log p_i \leq cn^{1-(\frac{1}{2})^i} n(\frac{1}{2})^i \log n \log \log n \\ \leq cn \log n \log \log n$$

If we let $r = \log \log n$, then we see that $p_r = \frac{n}{2}$, so we know half of the bits. A single second order Newton iteration extends this result to the full answer. Furthermore, we see that the total size for all r stages is $O(n \log n (\log \log n)^2)$.

Again (as in the polynomial reciprocal problem), we simply don't use all n bits for part A. If we let $N = \frac{n}{(\log n)^2}$, then performing the above algorithm on an N bit number produces a result accurate to N bits in size $O(N \log N (\log \log N)^2) = O(n)$.

The depth calculation is slightly more subtle. Looking at stage i , the depth of this stage is bounded by

$$c \log p_i + c \log k_i \log \log k_i \leq c \left(\frac{1}{2}\right)^i \log n \log \log n + c \log n$$

Summing over all r stages, and noticing that $\sum (\frac{1}{2})^i < 2$, we get a total depth of $O(\log n \log \log n)$. For the depth, decreasing the number of bits to N has no substantial effect, so the total depth is the same. \square

Now we look at part B of the reciprocal algorithm, namely using second order Newton iterations to extend the approximation of part A to n bits.

Theorem 6.2 Part B of the reciprocal algorithm produces the reciprocal accurate to n bits from the result of part A in size $O(M(n))$ and depth $O(\log n \log \log n)$.

PROOF: If n_1 bits are known initially, then after applying m second order Newton iterations, the approximation is extended to $n_2 = n_1 2^m$ bits. Using the number of bits produced by part A (theorem 6.1) as n_1 , letting $n_2 = n$, and solving for m , we get $m = 2 \log \log n$.

The size of second order Newton iteration on n_i bits is less than $cM(n_i)$ for some constant c . The number of bits in the last stage is n , and for simplicity of notation we number the stages from the end with $n_0 = n$ and $n_i = \frac{n_{i-1}}{2} = \frac{n}{2^i}$. The size of stage i is then less than $cM(\frac{n}{2^i})$, which is less than $\frac{c}{2^i} M(n)$ since $M(n)$ must be at least linear. The sum over all stages is now easily evaluated as $\sum cM(n_i) \leq 2cM(n)$, so the total size of part B is $O(M(n))$. The depth of each stage is $O(\log n)$, so the total depth of part B is $O(\log n \log \log n)$. \square

Now we are ready to put both parts together and state size and depth bounds for the entire reciprocal circuit.

Theorem 6.3 The reciprocal of an n bit number can be calculated to n -bit precision by a logspace uniform circuit in size $O(M(n))$ and depth $O(\log n \log \log n)$.

Theorem 6.3 is immediately applicable to other problems whose complexity is dominated by that of division. We define a *rational function* f as any function that can be written in the form $f(x) = p(x)/q(x)$, where p and q are fixed degree polynomials with coefficients that can be represented in fixed point binary with $O(n)$ bits. In a recent paper, H. Alt [2] shows how multiplication is simultaneous size and depth equivalent to the evaluation of polynomials; therefore, in particular, the evaluation of $p(x)$ and $q(x)$ above can be reduced to multiplication. These results can be combined with a single division to produce $f(x)$, which gives rise to the following corollary.

Corollary 6.1 Any rational function can be evaluated in $O(\log n \log \log n)$ depth and $O(M(n))$ size.

7 Conclusion and Open Problems

The important contribution of this paper is that the size bounds for multiplication are matched by a division circuit with depth less than $\Omega(\log^2 n)$; in fact, we match the best known depth bounds for logspace uniform reciprocal circuits while obtaining optimal size. Notice that if the size of multiplication (call this $M(n)$) is reduced, then using the new multiplication circuit in part B of our algorithm reduces the size of our division circuit to $O(M(n))$ also.

There are still interesting questions regarding the use of high order Newton iterations. We know that all rational functions can be evaluated in identical bounds (by Corollary 6.1). This gives strong evidence that other algebraic problems can be solved using this technique.

An open question remaining in integer division is reducing the depth of the logspace uniform circuits. This seems to be a very hard problem requiring a different approach entirely.

References

- [1] A. AHO, J. HOPCROFT, AND J. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.
- [2] H. ALT, *Comparing the Combinational Complexities of Arithmetic Functions*, JACM, Vol. 35, No. 2, April 1988, pp. 447-460.
- [3] P. W. BEAME, S. A. COOK, AND H. J. HOOVER, *Log Depth Circuits For Division And Related Problems*, 25th FOCS (1984), pp. 1-6.
- [4] A. BORODIN, *On Relating Time And Space To Size And Depth*, Siam J. Comput., Vol. 6, No. 4, December 1977, pp. 733-744.
- [5] S. A. COOK, Ph.D. thesis, Harvard Univ., Cambridge, MA, 1966.

- [6] J. HASTAD AND T. LEIGHTON, *Division in $O(\log n)$ Depth Using $O(n^{1+\epsilon})$ Processors*, unpublished note.
- [7] K. MELHORN AND F. P. PREPARATA, *Area-time Optimal Division for $T = \Omega((\log n)^{1+\epsilon})$* , Symp. on Theoretical Aspects of Computer Science (1986).
- [8] J. H. REIF, *Logarithmic Depth Circuits For Algebraic Functions*, 24th FOCS (1983), pp. 138-145.
- [9] J. H. REIF, *Logarithmic Depth Circuits For Algebraic Functions*, Siam J. Comput., Vol. 15, No. 1, Feb 1986, pp. 231-241.
- [10] J. H. REIF AND S. R. TATE, *Efficient Parallel Integer Division by High Order Newton Iteration*, preliminary draft, May 1988.
- [11] A. SCHÖNHAGE AND V. STRASSEN, *Schnelle Multiplikation grosser Zahlen*, Computing, 7 (1971), pp. 281-292.
- [12] N. SHANKAR AND V. RAMACHANDRAN, *Efficient Parallel Circuits And Algorithms For Division*, Info. Proc. Letters, 29 (1988), pp. 307-313.
- [13] J.F. TRAUB, *Iterative Methods For The Solution of Equations*, Chelsea Publishing Co., New York, N.Y., 1964.

Appendix A – Newton Iteration Complexity

A.1 – Description

At the heart of our circuit for high order Newton iteration is an improved modular powering algorithm based on [Reif, 86] and [Shankar and Ramachandran, 88]. Both algorithms use divide and conquer (by Discrete Fourier Transform) to work on powering problems with smaller numbers in parallel. For our division circuit, we desire a circuit for m th order Newton iteration of an n bit number that has size $O(nm^{O(1)}(\log n)^{O(1)})$. Unfortunately, the powering algorithm in [Reif, 86] has size that is quadratic in n , and the circuit of [Shankar and Ramachandran, 88], though it greatly improves the size bounds, also has a size that grows too fast in n for our intended application.

The divide and conquer approach of [Shankar and Ramachandran, 88] reduces the number of bits at each stage, but the power remains the same throughout. In our algorithm, we reduce *both* the number of bits and the power at each stage. Notice that if we want to raise an n bit number x to the m th power, where m is a perfect square, we can first raise m to the \sqrt{m} th power, and then raise this result to the \sqrt{m} th power. Unfortunately, we cannot guarantee that m is a perfect square, but we can let the first power be $p_1 = \lfloor \sqrt{m} \rfloor$. Next, see if $p_1(p_1 + 1) \leq m$, and if it is, the second power we take will be $p_2 = p_1 + 1$; if $p_1(p_1 + 1) > m$, then the second power will be just $p_2 = p_1$. The number x is first raised to the p_1 th power, and this result is then raised to the p_2 th power; the final result is $x^{p_1 p_2}$. Now since $p_1 p_2$ will usually not be m , we need to calculate an error term $e = m - p_1 p_2$. If we take x^e and multiply by the preceding result, we have the desired answer of x^m . A simple calculation shows that $e < \sqrt{m}$, so we have reduced the original powering problem to three smaller powerings, each of size $\approx \sqrt{m}$. Notice that the calculation of $x^{p_1 p_2}$ can happen in parallel with the calculation of x^e , so the depth is only that of two smaller powerings.

By reducing powers in this way and reducing the number of bits of each subproblem with discrete Fourier transforms, we reduce the size of our problem very quickly. For the depth bounds to work out as we needed, it was discovered that the number of bits should decrease faster than the powers. To achieve this, we only reduce the power half as often as we reduce the number of bits. We will consider a stage of reducing both power and bits followed by a second stage of reducing only the number of bits as a single level in our circuit. Notice that the stage of reducing only the number of bits is exactly the circuit of [Shankar and Ramachandran, 88].

The powering algorithm can be found in pseudocode in section A.2. The recursive call to MODPOWER_{SR} actually does a stage of the Shankar and Ramachandran circuit before recursively calling MODPOWER.

As shown in [Reif, 86] (also see [Shankar and Ramachandran, 88]), there will be no error with this algorithm as long as $2m(l + 1 + \log k) \leq k - 1$. It can be shown that for $m \geq 32$ and $n > m^2$ this inequality holds. It is also a simple check to see that at all levels of our algorithm, these inequalities hold.

Theorem 1 *The circuit that calculates MODPOWER has size $O(nm^4 \log n \log \log n)$, and depth $O(\log n + \log m \log \log m)$; furthermore, the circuit is logspace constructible.*

PROOF: We will use the notation $S(n, m)$ and $T(n, m)$ to denote the size and depth, respectively, of taking the m th power of an n bit number modulo $2^n + 1$ using our MODPOWER algorithm. We also use the variable c to denote an arbitrary constant; different occurrences of c do not necessarily imply that the values are the same. The MODPOWER algorithm deals only with integer values, and due to this, floors and ceilings are taken. Our analysis here is simplified by using exact values — a more careful analysis shows that we lose nothing by this simplification.

We first derive a recurrence equation for the size of the MODPOWER circuit. In the pseudocode, lines marked with an asterisk (*) take no size or depth in the circuit — they are calculated when the circuit is constructed. Ignoring the case of line (1) for now, we see that all lines other than those calling MPMACRO take size $O(M(n))$ and depth $O(\log n)$.

Deriving the size of MPMACRO is fairly straightforward. All steps except line (4) are easily done in size $O(k^2 \log k)$. Line (4) includes a stage of the Shankar and Ramachandran circuit as described in the text preceding the theorem. For each i , the size of this is bounded by $2\sqrt{km'}S(2\sqrt{km'}, m') + ck m' \log k$. As there are k different values of i for line (4), the total size of MPMACRO is bounded by $2k^{\frac{3}{2}}m'^{\frac{1}{2}}S(2k^{\frac{1}{2}}m'^{\frac{1}{2}}, m') + ck^2 m' \log k$.

Noticing that $m' \leq \lceil m^{\frac{1}{2}} \rceil$ and that MPMACRO is called three times, we get the following recurrence equation for the size of MODPOWER.

$$S(n, m) \leq 6n^{\frac{3}{4}}mS(2n^{\frac{1}{4}}m^{\frac{1}{2}}, m^{\frac{1}{2}}) + cnm^{\frac{3}{2}} \log n + cM(n) \quad (1)$$

To simplify our notation, we define several sequences of values, which denote the number of bits, the power to take, and the number of subproblems at level i .

Number of bits	$n_{i+1} = 2n_i^{\frac{1}{2}}m_i^{\frac{1}{2}}$	$n_0 = n$
Power	$m_{i+1} = m_i^{\frac{1}{2}}$	$m_0 = m$
Subproblems	$b_{i+1} = \prod_{j=0}^i 6n_j^{\frac{3}{4}}m_j$	$b_0 = 1$

The general form for these equations is easily found.

$$n_i = 2^{\frac{3}{4}[1-(\frac{1}{2})^i]}n(\frac{1}{2})^i m(\frac{1}{2})^{i-1}[1-(\frac{1}{2})^i]$$

$$m_i = m(\frac{1}{2})^i$$

$$b_i = 2^{i \log 12 - \frac{1}{4}[1 - (\frac{1}{4})^i]} n^{1 - (\frac{1}{4})^i} m^{3 - 5(\frac{1}{2})^i + 2(\frac{1}{4})^i}$$

Using these sequences, we can re-write equation (1) as

$$S(n, m) \leq b_i S(n_i, m_i) + c \sum_{j=0}^{i-1} b_j n_j m_j^{\frac{3}{2}} \log n_j + c \sum_{j=0}^{i-1} b_j M(n_j)$$

We will look at these terms one at a time. First of all, there are some cancellations in the $b_j n_j m_j^{\frac{3}{2}}$ term that simplify things; we can write $b_j n_j m_j^{\frac{3}{2}} = 2^{j \log 12} n m^{3 - \frac{3}{2}(\frac{1}{2})^j}$. We use the bound $M(n) \leq cn \log n \log \log n$ to see that $b_j M(n_j) \leq c 2^{j \log 12} n m^{3 - 3(\frac{1}{2})^j} \log n \log \log n$.

By the check before line (1), we never recurse more than $d = \log \log m$ levels; the number of bits at this stage is $n_d < cn(\frac{1}{4})^d$. The constant power circuit at this stage will take size $O(M(n_d)) = O((\frac{1}{4})^d n(\frac{1}{4})^d \log n \log \log n)$. We are now ready to solve the recurrence equation.

$$\begin{aligned} S(n, m) &\leq b_d S(n_d, m_d) + c n m^3 \sum_{j=0}^{d-1} 2^{j \log 12} m^{-\frac{3}{2}(\frac{1}{2})^j} + \\ &\quad c n m^3 \log n \log \log n \sum_{j=0}^{d-1} 2^{j \log 12} m^{-3(\frac{1}{2})^j} \\ &\leq b_d S(n_d, m_d) + c n m^3 (\log m)^4 + \\ &\quad c n m^3 (\log m)^4 \log n \log \log n \\ &\leq c n m^3 (\log m)^2 \log n \log \log n + \\ &\quad c n m^3 (\log m)^4 \log n \log \log n \\ &\leq c n m^3 (\log m)^4 \log n \log \log n \\ &\leq c n m^4 \log n \log \log n \end{aligned}$$

This agrees with the size bound stated in the theorem.

The depth of MPMACRO is even easier to compute than the size. The depth of all non-recursion lines is $O(\log k)$, and there is a single recursion for a total depth bound of $T(2k^{\frac{1}{2}} m'^{\frac{1}{2}}, m') + c \log k$.

Noticing that MPMACRO gets called twice sequentially (lines (4) and (5)), the total depth of MOD-POWER is

$$T(n, m) \leq 2T(2n^{\frac{1}{4}} m^{\frac{1}{2}}, m^{\frac{1}{2}}) + c \log n \quad (2)$$

Re-writing this in terms of the sequences we defined above, we get

$$T(n, m) \leq 2^i T(n_i, m_i) + c \sum_{j=0}^{i-1} 2^j \log n_j$$

As noted before, we never recurse more than $d = \log \log m$ levels. The final stage takes depth $T(n_d, m_d) \leq c(\frac{1}{4})^d \log n$, so substituting this and ex-

panding $\log n_j$, we get the new equation

$$\begin{aligned} T(n, m) &\leq c 2^d \left(\frac{1}{4}\right)^d \log n + c \sum_{j=0}^{d-1} 2^j \frac{4}{3} + \\ &\quad c \sum_{j=0}^{d-1} 2^j \left(\frac{1}{4}\right)^j \log n + c \sum_{j=0}^{d-1} 2^j \left(\frac{1}{2}\right)^j \log m \\ &\leq c \log n + c \log m + c \log n + c \log m \log \log m \\ &\leq c \log n + c \log m \log \log m \end{aligned}$$

The last line gives the depth bound stated in the theorem. \square

Corollary 1 *We can calculate x^m , where x is an n bit number, in size $O(nm^5 \log n \log \log n)$, and depth $O(\log n + \log m \log \log m)$. This circuit is also logspace constructible.*

PROOF: Simply use the modular powering algorithm of Theorem 1 to calculate $x^m \bmod 2^{nm} + 1$. This ring is large enough to hold the exact answer, so the modular result will be the same as the exact result. \square

Corollary 2 *We can calculate the m th order Newton iteration on an n bit number with a logspace uniform circuit of size $O(nm^6 \log n \log \log n)$, and depth $O(\log n + \log m \log \log m)$.*

PROOF: Looking at the Newton iteration formula of section 4, we first need to calculate $u = 1 - xy_i$. This can easily be done in $O(M(n))$ size and $O(\log n)$ depth. Next, we need to calculate u^i for $0 \leq i < m$, which is done by the above circuit. The powers of u are then all added together with size $O(nm)$ and depth $O(\log n)$, and the final multiplication by y_i is performed. Clearly, the cost of performing the m powerings dominates the entire circuit, so the total size is $O(nm^6 \log n \log \log n)$, and the depth is $O(\log n + \log m \log \log m)$. \square

When this powering algorithm is used to compute reciprocals (either by the methods of [Melhorn and Preparata, 86] and [Shankar and Ramachandran, 88] or by the high order Newton iteration of [Reif and Tate, 88]), the finite approximation to the power series can be factored so that the greatest power taken is m^ϵ for some constant $\epsilon > 0$. For the non-iterative reciprocal algorithms, this simply gives another $O(\frac{1}{\epsilon} n^{1+\epsilon})$ size, $O(\log n \log \log n)$ depth algorithm. However, if we look at the high order Newton iteration formula and set $\epsilon = \frac{1}{6}$, we have the following corollary.

Corollary 3 *We can calculate the m th order Newton iteration on an n bit number with a logspace uniform circuit of size $O(nm \log n \log \log n)$, and depth $O(\log n + \log m \log \log m)$.*

A.2 – Pseudocode for MODPOWER

```

MODPOWER( $x, m, n$ )    /* Calculate  $x^m \bmod 2^n + 1$  */
  if  $m < 32$  then
    (1) Perform constant power calculation using Schonhage-Strassen algorithm.
  else
    (*)  $k \leftarrow \lceil \sqrt{nm} \rceil$ 
    (*)  $l \leftarrow \lfloor \frac{n}{k} \rfloor$ 
    (*)  $p_1 \leftarrow \lfloor \sqrt{m} \rfloor$ 
    (*) if  $p_1(p_1 + 1) \leq m$  then
    (*)    $p_2 \leftarrow p_1 + 1$ 
    (*) else
    (*)    $p_2 \leftarrow p_1$ 
    (*) fi
    (*)  $e \leftarrow m - p_1 p_2$ 

    In parallel do part1, part2
      part1:
    (2)    $t \leftarrow \text{MPMACRO}(x, e)$ 
      part2:
    (3)    $y \leftarrow \text{MPMACRO}(x, p_1)$ 
    (4)    $z \leftarrow \text{MPMACRO}(y, p_2)$ 
    od
    (5)   MODPOWER  $\leftarrow zt \bmod 2^n + 1$ 
  fi

 $y' \leftarrow \text{MPMACRO}(x', m')$  /* Uses  $k, l$ , and  $n$  from above */
 $y' \leftarrow x'$ 

(1) Divide  $y'$  into  $k$  blocks of  $l$  bits each, such that  $y' = \sum_{i=0}^{k-1} y_i 2^{il}$  and  $0 \leq y_i < 2^l$  for all  $i$ 
(2)  $(y_0, y_1, y_2, \dots, y_{k-1}) \leftarrow (y_0, 2^1 y_1, 2^2 y_2, \dots, 2^{k-1} y_{k-1}) \bmod 2^k + 1$ 
(3)  $(y_0, y_1, y_2, \dots, y_{k-1}) \leftarrow \text{DFT}_k(y_0, y_1, y_2, \dots, y_{k-1}) \bmod 2^k + 1$ 
    In parallel for  $i = 0, 1, \dots, k-1$  do
    (4)    $y_i \leftarrow \text{MODPOWER}_{\text{SR}}(y_i, m', k)$  /* Uses [Shankar and Ramachandran, 88] */
    od
    (5)  $(y_0, y_1, y_2, \dots, y_{k-1}) \leftarrow \text{DFT}_k^{-1}(y_0, y_1, y_2, \dots, y_{k-1}) \bmod 2^k + 1$ 
    (6)  $(y_0, y_1, y_2, \dots, y_{k-1}) \leftarrow (y_0, 2^{-1} y_1, 2^{-2} y_2, \dots, 2^{-(k-1)} y_{k-1}) \bmod 2^k + 1$ 
    (7)  $y' \leftarrow y_0 + y_1 2^l + y_2 2^{2l} + \dots + y_{k-1} 2^{(k-1)l} \bmod 2^n + 1$ 

```