OPTIMAL SIZE INTEGER DIVISION CIRCUITS*

JOHN H. REIF† AND STEPHEN R. TATE†

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 $\Omega(\log^2 n)$. The circuit families described in this paper are logspace uniform; that is, they can be constructed by a deterministic Turing machine in space $O(\log n)$.

The 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, this algorithm implies that any rational function can be evaluated in these complexity bounds.

As an introduction to high-order iterative methods a circuit is first presented 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.

Key words. algebraic computation, integer division, circuit complexity, powering

AMS(MOS) subject classifications. 68Q25, 68Q40

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 noninput 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, a circuit can be viewed as computing 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 two 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

* Received by the editors September 28, 1988; accepted for publication (in revised form) January 24, 1990. 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 the Defense Advanced Research Projects Agency contract DAAL03-88-K-0195, and by the Air Force Office of Scientific Research contract AFOSR-87-0386.

† Department of Computer Science, Duke University, Durham, North Carolina 27706.

computation. In § 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 $SM(n) = O(n \log n \log\log n)$ and $M(n) = O(n \log n \log\log n)$.

The problem of integer division was examined by 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.

A long-standing open question has been 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 Reif presented a logspace uniform circuit based on wrapped convolutions with depth $O(\log n(\log\log n)^2)$ and slightly more than polynomial size [8]. A year later Beame, Cook, and Hoover presented a polynomial time uniform circuit based on Chinese remaindering with polynomial size and depth $O(\log n)$ [3]. A revised paper by Reif reduced the depth bounds on the logspace uniform circuit to $O(\log n \log\log n)$ while simultaneously achieving polynomial size [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 using 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 $O(n^{1+\epsilon}/\epsilon^4)$, 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 $O((1/\epsilon^2)\log n)$, 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 $O(\log n \log\log n)$. Newton approximations of high degree are used to gain as many bits as possible in the early stages, and thus reduce the overall number of stages required. An important property of the new algorithm is 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 high-order iterative methods. 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. Our algorithm consists of 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 kth 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, k can be made large, so much more work can be done on each stage than simply doubling the number of bits as done by Cook.

The value of k for a particular stage is selected by making the size of every stage meet the same bound. The result is 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. High-order iterations for polynomial inverse. Let $R = \{D, +, \cdot, 0, 1\}$ be an arbitrary ring; 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. The degree $k-1$ polynomial derived in this way from $p(x)$ is denoted by $p_k(x) = \sum_{i=0}^{k-1} a_{n-k+i} x^i$. In problems 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. Note 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

$$q(x) = \mathrm{RECIPROCAL}(p(x)) = \left\lfloor \frac{x^{2n-2}}{p(x)} \right\rfloor .^{1} \tag{1}$$

It is easy to see that $q(x)$ must have degree $n-1$.

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 $\mathrm{RECIPROCAL}(p_d(x))$ as the length d "estimate" — note that $\mathrm{RECIPROCAL}(p(x))$ can be written as $\mathrm{RECIPROCAL}(p_n(x))$. To produce the estimate of length kd, first calculate the intermediate polynomial

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

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

$$q(x) = \left\lfloor \frac{r(x)}{x^{(k-1)(kd-2)}} \right\rfloor . \tag{3}$$

LEMMA 3.1. *Given $s(x) = \mathrm{RECIPROCAL}(p_d(x))$, the polynomial $q(x)$ computed from (3) is exactly $\mathrm{RECIPROCAL}(p_{kd}(x))$; furthermore, $q(x)$ can be computed in $O(k^3 d \log(kd))$ size and $O(\log(kd))$ depth.*

Proof. First we prove the correctness of the iteration formula (3). The lemma can be stated in a different (but equivalent) way; that is, that (3) produces a polynomial $q(x)$ such that $q(x)p_{kd}(x) = x^{2kd-2} + t(x)$ 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 $\mathrm{degree}[t_1(x)] < d-1$.

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

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

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

$$p_{kd}(x)q(x) = x^{2kd-2} - \left\lfloor \frac{\left[-(x^{(k-1)d}t_1(x) + s(x)p'(x))\right]^k}{x^{(k-1)(kd-2)}} \right\rfloor .$$

[1] The floor function for the division of polynomials is analogous to the floor function applied to integers. In other words, in (1), $q(x)$ is the unique polynomial such that $x^{2n-2} = q(x)p(x) + r(x)$, where $r(x)$ is the remainder and satisfies $\mathrm{degree}[r(x)] < \mathrm{degree}[p(x)]$.

To simplify notation, let

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

so $p_{kd}(x)q(x) = x^{2kd-2} + t(x)$.

Examining the degrees of the components of $t(x)$ we see that the numerator is just $g(x)$ which, on closer examination, satisfies degree$[g(x)] \leq kd - 2$. After the powering of $g(x)$ we see that degree$[g(x)]^k \leq k^2d - 2k$. The division gives $t(x)$ with 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 kth 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, and 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 (3) to get the complete polynomial reciprocal algorithm. The details are described in the proof of the following theorem.

THEOREM 3.2. *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. Let $f(i) = \lceil m(1 - (2/3)^{i-1}) \rceil$; now we can define a sequence of values by $d_i = 2^{f(i)}$. Note 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 $k_i = 2^{f(i+1)-f(i)}$, and it is easy to show that $f(i+1) - f(i) \leq (m/3)\,(2/3)^{i-1} + 1$. 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 — it can be verified by solving $m(1 - (2/3)^{i-1}) = m - 1$), so the total size of all stages of this algorithm is $O(n \log n \log\log n)$, and the total depth is $O(\log n \log\log n)$.

However, the algorithm that was just described is not quite what we use. If we take the first $n/\log^2 n$ coefficients of $p(x)$ and find the reciprocal of the polynomial defined by these coefficients, then letting $n' = n/\log^2 n$ the previously mentioned algorithm takes 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 (using the result of part A as an initial estimate), 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. Calculating integer powers. At the heart of our circuit for integer reciprocals is an improved modular powering algorithm based on previous results of Reif [9], and Shankar and Ramachandran [12]. Both previous algorithms use divide and conquer (by Discrete Fourier Transform) to work on powering problems with smaller numbers in parallel. For our division circuit, we need a circuit for mth 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 [9], has size that is quadratic in n, and the circuit of

Shankar and Ramachandran [12], though it improves the size bounds, also has size that grows too fast in n for our intended application.

The divide-and-conquer approach of Shankar and Ramachandran [12], 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. Note that to raise an n bit number x to the mth power, where m is a perfect square, we can first raise x to the $\sqrt{m}$th power, and then raise this result to the $\sqrt{m}$th power. Unfortunately, m is often not a perfect square, so 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_1th power, and this result is then raised to the p_2th power; the final result is $x^{p_1p_2}$. Now since p_1p_2 will usually not be m, we need to calculate an error term $e = m - p_1p_2$. If we take x^e and multiply by the preceding result, the result is the desired answer of x^m. A simple calculation shows that $e < \sqrt{m}$, so the original powering problem has been reduced to three smaller powerings, each of size $\approx \sqrt{m}$. Note that the calculation of $x^{p_1p_2}$ can happen in parallel with the calculation of x^e, so the depth is only that of two smaller powerings (not three).

By reducing powers in this way and reducing the number of bits of each subproblem with discrete Fourier transforms, the size of the problem is reduced 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, the power is reduced only half as often as 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.

The powering algorithm can be found in pseudocode in Fig. 1. The recursive call to $\mathrm{MODPOWER}_{\mathrm{SR}}$ actually does a stage of the Shankar and Ramachandran circuit before recursively calling MODPOWER.

As shown in Reif [9], (also see Shankar and Ramachandran [12]), there will be no error with this algorithm as long as $2m(l+1+\log k) \leq k-1$, which is satisfied whenever $m \geq 32$ and $n > m^2$. A simple check shows that at all levels of our algorithm, these inequalities hold.

THEOREM 4.1. *The circuit that calculates* $\mathrm{MODPOWER}(\cdot, m, n)$ *with the constraint* $m \leq \sqrt{n}$ *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 mth power of an n bit number modulo 2^n+1 using our MODPOWER algorithm. The MODPOWER algorithm deals only with integer values, and consequently, floors and ceilings are often taken. These are analyzed by repeatedly applying the following inequality — when bounding a product such as $\lceil m\rceil\lceil n\rceil$, note that

$$\lceil m\rceil\lceil n\rceil < (m+1)(n+1) < mn\left(1 + \frac{1}{m} + \frac{1}{n} + \frac{1}{mn}\right). \tag{4}$$

If there is a constant lower bound for m and n, then $\lceil m\rceil\lceil n\rceil$ can be bounded by $\lceil m\rceil\lceil n\rceil < cmn$ for some constant c (in many cases below, we actually bound the constant c).

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

MODPOWER(x, m, n) /* Calculate $x^m \bmod 2^n + 1$ */
    if $m < 32$ **then**
(1)   Calculate using Schönhage-Strassen multiplication 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) $t \leftarrow$ MPMACRO(x, e)
          part2:
(3) $y \leftarrow$ MPMACRO(x, p_1)
(4) $z \leftarrow$ MPMACRO(y, p_2)
        od
(5)   MODPOWER $\leftarrow zt \bmod 2^n + 1$
    fi

$y' \leftarrow$ 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
        for all i, $0 \leq y_i < 2^l$
(2) $(y_0, y_1, y_2, \cdots, y_{k-1}) \leftarrow (y_0, 2^1 y_1, 2^2 y_2, \cdots, 2^{k-1} y_{k-1}) \bmod 2^k + 1$
(3) $(y_0, y_1, y_2, \cdots, y_{k-1}) \leftarrow \mathrm{DFT}_k(y_0, y_1, y_2, \cdots, y_{k-1}) \bmod 2^k + 1$
    In parallel for $i = 0, 1, \cdots, k - 1$ **do**
(4) $y_i \leftarrow$ MODPOWER$_{\mathrm{SR}}(y_i, m', k)$ /* Uses ([12]) */
    od
(5) $(y_0, y_1, y_2, \cdots, y_{k-1}) \leftarrow \mathrm{DFT}_k^{-1}(y_0, y_1, y_2, \cdots, y_{k-1}) \bmod 2^k + 1$
(6) $(y_0, y_1, y_2, \cdots, y_{k-1}) \leftarrow (y_0, 2^{-1} y_1, 2^{-2} y_2, \cdots, 2^{-(k-1)} y_{k-1}) \bmod 2^k + 1$
(7) $y' \leftarrow y_0 + y_1 2^l + y_2 2^{2l} + \cdots + y_{k-1} 2^{(k-1)l} \bmod 2^n + 1$

FIG. 1. *Pseudocode for* MODPOWER.

— 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 total size $O(M(n))$ and total depth $O(\log n)$.

Deriving the size of MPMACRO can be done as follows. Assuming that $m > 32$ (so there is at least one level of recursion), let $k_1 = \lceil \sqrt{nm} \rceil$ be the k from MODPOWER, and let $k_2 = \lceil 2\sqrt{k_1 m'} \rceil$ be the k from the application of the Shankar and Ramachandran circuit. All steps except line (4) are easily done in size $O(k_1^2 \log k_1)$. 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 reduction is bounded by $k_2 S(k_2, m') + ck_2^2 \log k_2$ for some constant c. As there are k_1 different values of i for line (4), the total size of MPMACRO is bounded by $k_1 k_2 S(k_2, m') + ck_1 k_2^2 \log k_2$.

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

$$S(n,m) = \begin{cases} c_1 M(n) & \text{if } m \leq 32, \\ 3k_1 k_2 S(k_2, \lceil m^{1/2} \rceil) + c_2 k_1 k_2^2 \log k_2 + c_3 M(n) & \text{otherwise.} \end{cases}$$

The claim is that for some constant c, $S(n,m) \leq cnm^4 \log n \log\log n$ satisfies this for all $m \leq \sqrt{n}$. For $m \leq 32$, this is obviously true.

For $m > 32$ but $\sqrt{m} \leq 32$, the recursive cost is given by the top line of the recurrence equation. Therefore, the total size is bounded by $3k_1 c_1 M(k_1) + c_2 k_1 \log k_1 + c_3 M(n)$. But $k_1 < n$ and $3c_1 k_1^2 < c_4 mn$ for some constant c_4, so this is bounded by $(c_1 + c_3)M(n) + c_2 n \log n$ — and it follows that the size claim holds.

For $\sqrt{m} > 32$, we need to look more closely at k_1 and k_2. Expanding k_2, we see that $k_2 = \left\lceil 2 \left(\lceil \sqrt{nm} \rceil \lceil \sqrt{m} \rceil \right)^{1/2} \right\rceil$. Using the technique above for bounding products of ceilings, we bound $k_2 < \lceil 2.04 n^{1/4} m^{1/2} \rceil$. Using the same method, we see that $k_1 k_2 < 2.05 n^{3/4} m$. Using these facts, if $\sqrt{m} > 32$, then

$$S(n,m) \leq 6.15 n^{3/4} m S(\left\lceil 2.04 n^{1/4} m^{1/2} \right\rceil, \left\lceil m^{1/2} \right\rceil) + c_5 n m^{3/2} \log n + c_3 M(n).$$

Using our claim on the right-hand side and repeatedly using the bound from equation (4) for bounding products of ceilings, the size claim can be proved.

The depth of MPMACRO is even easier to compute than the size. The depth of all nonrecursion lines is $O(\log k_1)$, and there is a single recursion for a total depth bound of $T(k_2, m') + c \log k_1$.

Noting that MPMACRO gets called twice sequentially (lines (4) and (5)), the total depth of MODPOWER is bounded by $2T(k_2, m') + c \log n$. Using the bounding equations calculated above, the recurrence equations for the depth are

$$T(n,m) = \begin{cases} c_1 \log n & \text{if } m \leq 32, \\ 2T(\lceil 2.04 n^{1/4} m^{1/2} \rceil, \lceil m^{1/2} \rceil) + c_2 \log n & \text{otherwise.} \end{cases}$$

Our claim is that for some constant c, $T(n,m) \leq c(\log n + \log m \log\log m)$ satisfies the above equation. Again, if $m \leq 32$, there is nothing to prove.

If $m > 32$ but $\sqrt{m} \leq 32$, then there is just the one recursive call as in the size analysis. Since $\log k_2 < \log n$, the recurrence equation for the depth becomes $T(n,m) \leq (2 + c_2) \log n$ — therefore, setting $c = 2 + c_2$ is sufficient to prove the claim.

If $\sqrt{m} > 32$, it is important to note the following two inequalities:

$$\log \left\lceil 2.04 n^{1/4} m^{1/2} \right\rceil < \tfrac{1}{4} \log n + \tfrac{1}{2} \log m + 1.2,$$

$$\log \left\lceil m^{1/2} \right\rceil \log\log \left\lceil m^{1/2} \right\rceil < \tfrac{1}{2} \log m \log\log m + 0.06 \log\log m - 0.48 \log m - 0.05.$$

Using these values to put the claim in the right-hand side of the recurrence equations results in a proof that the claim holds for $c = 5c_2$.

As for the circuit being logspace constructible, it should be noted that all calculations made in the construction of the circuit (the lines marked with (*) in Fig. 1) deal with numbers that are $O(\log n)$ bits long. In other words, these calculations only need to be done in space *linear* in the length of the numbers used — this is, of course, easily done. $\square$

COROLLARY 4.2. MODPOWER *can compute* x^m, *where* x *is an* n *bit number and* $m \leq \sqrt{n}$, *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 4.1 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. $\square$

5. High-order iteration for integer division. The following definition is useful when describing the amount of error present in an approximation.

DEFINITION. 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 $\frac{1}{2} < x \leq 1$, 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 two (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 of the form "find x such that $f(x) = 0$" for some given function f. 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 kth order Newton iteration for the reciprocal problem is given by

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

where the values y_i are the approximations to y.

In the following error analysis, 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 5.1. *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 $x \leq 1$, this 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 5.2. *If k is even, then after applying a kth degree Newton iteration at step i, $y_{i+1} \leq y$.*

In the discussion above, we assumed that calculations were performed 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 us 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 $\tilde{y} \geq y$. Let $\epsilon_{\tilde{x}} = x - \tilde{x}$, and $\epsilon_{\tilde{y}} = \tilde{y} - y$.

LEMMA 5.3. *If $|\epsilon_{y,i}| \leq c \leq \frac{1}{4}$ for some value c, and can insure that $\epsilon_{\tilde{y}} \leq c^k$, then performing the kth order iteration (k even) using $\tilde{y}$ will result in $|\epsilon_{y,i+1}| \leq c^k$.*

Proof. It is important to note that we are doing the exact Newton iteration for $\tilde{y}$. There are three cases to consider, 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 5.4. *If y_i is accurate to p bits with $p \geq 2$, then applying a kth 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 note 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 5.3 directly applies to give $|\epsilon_{y,i+1}| \leq c^k = 2^{-kp}$, so y_{i+1} is accurate to kp bits. □

6. 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 kth 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 kth order Newton iteration, using pk bits, then see what happens when two more bits are used.

To calculate the required approximations, we use the new method of powering introduced in § 4 to obtain the following results.

THEOREM 6.1. *The kth order Newton iteration of a p bit number (using pk bit calculations and giving a pk bit result) can be computed by a logspace uniform circuit family of size $O(pk^7 \log pk \log\log pk)$, and depth $O(\log p + \log k \log\log k)$.*

Proof. Looking at the Newton iteration formula of § 5, we first need to calculate $u = 1 - xy_i$. This can easily be done in $O(M(pk))$ size and $O(\log pk)$ depth. Next, we need to calculate u^i for $0 \leq i < k$, which is done by the circuit of § 4. The powers of u are then all added together with size $O(pk^2)$ and depth $O(\log pk)$, and the final multiplication by y_i is performed. Clearly, the cost of performing the k powerings dominates the entire circuit, so the total size is $O(pk^7 \log pk \log\log pk)$, and the depth is $O(\log pk + \log k \log\log k) = O(\log p + \log k \log\log k)$. □

It is important to note that the summation in the Newton iteration formula is a simple truncated power series and can be factored in exactly the same manner as the reciprocal power series in Melhorn and Preparata [7] and Shankar and Ramachandran [12]. After such a factoring, the largest power that needs to be taken is k^ϵ for some constant $\epsilon > 0$, and the resulting circuit has size $O(pk^{1+6\epsilon} \log pk \log\log pk)$ while the depth remains essentially unchanged. Setting $\epsilon = \frac{1}{6}$, we get the following corollary.

COROLLARY 6.2. *The kth order Newton iteration of a p bit number (using pk bit calculations) can be calculated by a logspace uniform circuit family of size $O(pk^2 \log pk \log\log pk)$ 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 and 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, Corollary 6.2 and Theorem 5.4, we get the following corollary.

COROLLARY 6.3. *An approximation accurate to pk bits can be obtained from a p bit approximation by a logspace uniform circuit of size $O(pk^2 \log pk \log\log pk)$ and depth $O(\log p + \log k \log\log k)$.*

7. 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. Let p_i denote the number of bits of accuracy at stage i, and define a sequence of accuracies by $p_i = n^{1-(1/2)^i}$; note that $p_0 = 1$ (only one bit needs to be known initially).

THEOREM 7.1. *Part* A *of the reciprocal algorithm calculates the reciprocal of x accurate to $n/(\log n)^2$ 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 is needed at stage i — call this k_i:

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

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

$$\begin{aligned} cp_i k_i^2 \log p_i k_i \log\log p_i k_i &\leq cn^{1-(1/2)^i} n^{(1/2)^i} \log n \log\log n \\ &\leq cn \log n \log\log n. \end{aligned}$$

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. Therefore, 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 do not use all n bits for part A. If we let $N = n/(\log n)^2$, 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

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

Summing over all r stages, and noting that $\sum \left(\frac{1}{2}\right)^i$ is bounded by a constant (it is bounded by 2, to be exact), the total depth is $O(\log n \log\log n)$. For the depth, decreasing the number of bits to N has no substantial effect, so the total depth is the same. ☐

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 7.2. *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 7.1) 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 $n_i = n_{i-1}/2 = n/2^i$. The size of stage i is then less than $cM(n/2^i)$, which is less than $(c/2^i)M(n)$ 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 $O(\log n \log\log n)$. ☐

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

THEOREM 7.3. *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 7.3 is immediately applicable to other problems whose complexity is dominated by that of division. A *rational function* f is 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, 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 7.4. *Any rational function can be evaluated in $O(\log n \log\log n)$ depth and $O(M(n))$ size.*

8. 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. Note 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 7.4). 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. V. AHO, J. E. HOPCROFT, AND J. D. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[2] H. ALT, *Comparing the combinatorial complexities of arithmetic functions*, J. Assoc. Comput. Mach., 35 (1988), pp. 447–460.

[3] P. W. BEAME, S. A. COOK, AND H. J. HOOVER, *Log depth circuits for division and related problems*, SIAM J. Comput., 15 (1986), pp. 994–1003.

[4] A. BORODIN, *On relating time and space to size and depth*, SIAM J. Comput., 6 (1977), pp. 733–744.

[5] S. A. COOK, *On The Minimum Computation Time of Functions*, Ph.D. thesis, Harvard University, Cambridge, MA, 1966.

[6] J. HASTAD AND T. LEIGHTON, *Division in $O(\log n)$ depth using $O(n^{1+\epsilon})$ processors*, unpublished note, 1986.

[7] K. MELHORN AND F. P. PREPARATA, *Area-time optimal division for $T = \Omega((\log n)^{1+\epsilon})$*, in Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Computer Science 210, Springer-Verlag, New York, 1986, pp. 341–352.

[8] J. H. REIF, *Logarithmic depth circuits for algebraic functions*, in Proc. 24th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Washington, D.C., 1983, pp. 138–145.

[9] ———, *Logarithmic depth circuits for algebraic functions*, SIAM J. Comput., 15 (1986), pp. 231–241.

[10] J. H. REIF AND S. R. TATE, *Efficient parallel integer division by high order newton iteration*, preliminary draft, 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*, Inform. Process. Lett, 29 (1988), pp. 307–313.

[13] J. F. Traub, *Iterative Methods for The Solution of Equations*, Chelsea, New York, 1964.