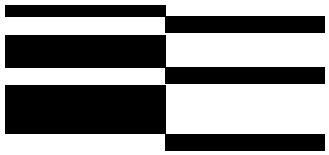
Part IV

Fundamental Parallel Algebraic Algorithms

1



Newton Iteration and Integer Division

Stephen R. Tate
Department of Computer Science
University of North Texas
P.O. Box 13886
Denton, TX 76203–6886
srt@cs.unt.edu

1.1 Introduction

At the heart of all numerical computations are the basic operations of addition, subtraction, multiplication, and division (also, to a lesser extent, more complex operations such as powering, finding square roots, computing logarithms, etc.). It is vital to study these problems from the standpoint of parallel algorithms, because even in commonly used single processor machines the basic operations are done in parallel (by parallel paths through low-level logic circuits).

Early in school, a student learns how to perform the functions of addition, subtraction, multiplication, and division. In fact, these topics are usually presented in this order due to the increasing difficulty of the operations. Studies in parallel algorithms support the sense of difficulty assigned by our elementary school teachers—optimal algorithms exist for addition and subtraction, while good algorithms exist for multiplication (even the best of which is not known to be optimal), and division seems to be even harder. For more information on the operations of addition, subtraction, and multiplication, see the references section at the end of this chapter.

The common computational model used when examining arithmetic problems is the *bounded fan-in Boolean circuit*. A bounded fan-in circuit is simply a directed acyclic graph with each node having bounded in-degree. There are two sets of special nodes: the input nodes (which have in-degree zero) and the output nodes (which have out-degree zero). Furthermore, since the circuits we consider are Boolean, each non-input node is labeled with a Boolean function (AND, OR, or NOT). By applying Boolean values to the input nodes and computing all nodes as the labeled function from their predecessor values, circuits can be regarded as computing functions. To compute a function, *circuit families* are considered (one circuit for each possible input size). The *size* of a circuit family is a function $S(n)$ that gives (for each $n \geq 1$) the number of nodes in the circuit for inputs of length n. The *depth* of a circuit family is a function $D(n)$ that gives (for each $n \geq 1$) the length of the longest path from an input node to an output node in the circuit for length n inputs.

In this chapter, the parallel complexity of division is compared with the complexity of the other elementary operations. The problem of integer division is defined to be a function that takes a pair of input values (y, x), and produces the pair of values (q, r) such that $y = qx + r$, where $0 \leq r < x$ (i.e., a quotient and a remainder). Reduction of division to multiplication

via Newton approximation is shown to provide good sequential results, but these results do not translate well to parallel algorithms. In this chapter, we describe a parallel algorithm due to Reif and Tate [14] which is a modified version of Newton approximation (called *high order* Newton approximation for reasons that will become clear). This algorithm obtains parallel results that are almost optimal. On the way to the results for division, it will be discovered that finding limited integer powers is vital for division, so ways of accomplishing this are discussed.

As is standard practice when comparing the complexity of algorithms, the focus of this chapter will be on reductions to other problems. It is sufficient to consider the problem of finding reciprocals in place of division. As we are considering only integer operations, the idea of a reciprocal is not clear—in general, the real reciprocal of an integer will *not* be an integer. Therefore, given an n-bit input integer x, the integer reciprocal is defined as the value

$$\left\lfloor \frac{2^{2n}}{x} \right\rfloor.$$

Notice that this is simply the shifted binary fixed point approximation to the real reciprocal; it should be obvious how the reciprocal can be used with a constant number of multiplications to solve the division problem.

The notation $\leq_{sd}$ denotes a constant size and depth reduction; in other words, if f and g are two functions, then $f \leq_{sd} g$ if, given any circuit family computing g in size $S(n)$ and depth $D(n)$, a circuit family can be constructed which computes f in size $O(S(n))$ and depth $O(D(n))$. Letting SQ denote the function that squares an n-bit integer and MULT denote the problem of multiplying two n-bit integers, it is easy to see that SQ $\leq_{\text{sd}}$ MULT. It is also true, but not quite as obvious, that MULT $\leq_{sd}$ SQ since $xy = \frac{1}{2}[(x+y)^2 - x^2 - y^2]$ (addition is easily accomplished, and the multiplication by $\frac{1}{2}$ is simply a binary shift by one bit). The notation $\equiv_{sd}$ is used for two problems that are constant size-depth reducible to each other, so SQ $\equiv_{sd}$ MULT as just shown.

Re-examining our rather arbitrary hierarchy of difficulty for arithmetic problems, a good candidate for a reduction of division would be multiplication. In fact, letting REC denote the integer reciprocal problem and using the new notation, it can be shown that SQ $\leq_{sd}$ REC by

$$x^2 = \frac{1}{\frac{1}{x} - \frac{1}{x+1}} - x.^1 \tag{1.1}$$

[1]This is actually an abuse of notation, since equation (1.1) uses real reciprocals instead of the defined integer reciprocal; however, it is not hard to see how the integer reciprocal can be used to approximate real reciprocals in the calculation of equation (1.1).

Noting that the $\leq_{sd}$ relation is transitive, this also means that MULT $\leq_{sd}$ REC, so finding reciprocals is at least as hard (in the sense of constant size-depth reductions) as multiplication. This verifies the fact that multiplication is a good candidate when trying to reduce division.

Throughout this chapter, the notation $M(n)$ will be used to represent the smallest size required by any circuit family that multiplies two n-bit numbers in $O(\log n)$ depth. As there are no known optimal algorithms for multiplication at this time, the exact value of $M(n)$ is unknown; however, the value is easily lower-bounded by $M(n) = \Omega(n)$ and upper-bounded by $M(n) = O(n \log n \log\log n)$ (the upper bound is due to an algorithm by Schönhage and Strassen—see the references at the end of the chapter for more information). It is assumed that $M(n)$ satisfies the equation

$$M(cn) \leq cM(n) \tag{1.2}$$

for all positive $c \leq 1$. Almost all complexity measures that are $\Omega(n)$ satisfy this bound, so the assumption is not too great.

In the text that follows, the notation $\mathsf{RECIPROCAL}(x, n)$ refers to the *function* of integer reciprocal, without reference to a particular algorithm; the arguments x and n denote the input value and the size of the input, respectively. When referring to specific algorithms that compute the reciprocal function, the notation used will be $\mathsf{RECIP1}(x, n)$, $\mathsf{RECIP2}(x, n)$, etc.

1.2 Newton Approximation

Newton approximation is a tool commonly used by numerical analysts to find the zeros of a function. In numerical analysis terms, Newton approximation (in general) has quadratic convergence—what this means to the division problem will become clear shortly.

Consider a differentiable function $f(x)$ that has first derivative $f'(x)$ and has a zero at x_0 (so $f(x_0) = 0$). Assuming that $f'(x)$ is non-zero in a reasonable neighborhood of x_0, we can make an initial guess for x_0 (call the initial guess y_1) and use the slope $f'(y_1)$ to estimate how far y_1 is from the zero. This produces a new estimate for x_0 (call it y_2) and the process can be repeated producing a sequence of estimates $y_1, y_2, y_3, \ldots$ that converges to x_0 for all well-behaved functions and good initial approximations. In mathematical terms, this becomes

$$y_{i+1} = y_i - \frac{f(y_i)}{f'(y_i)}. \tag{1.3}$$

The convergence rate for the general case is beyond the scope of interest of this chapter—the interested reader can consult any introductory numerical analysis text.

Consider the function $f(y) = 1 - \frac{1}{xy}$. Obviously, $\frac{1}{x}$ is a zero of f, and the derivative $f'(y) = \frac{1}{xy^2}$ is non-zero for all $y \neq 0$. Using this function f, equation (1.3) gives a sequence defined by

$$y_{i+1} = 2y_i - xy_i^2. \tag{1.4}$$

This equation will take a good initial estimate and converge to $\frac{1}{x}$. A word of warning is appropriate here—notice how easily we slipped into solving the problem of real reciprocals instead of integer reciprocals. Fortunately, the problem is not too great—as was noted before, the integer reciprocal is simply a scaled representation of the fixed point binary approximation to the real reciprocal. Re-writing the above equation with this scaling in mind, the following equation generates a sequence that converges to the integer reciprocal using only integer operations.

$$y_{i+1} = \left\lfloor \frac{2^{2n+1}y_i - xy_i^2}{2^{2n}} \right\rfloor \tag{1.5}$$

This formula works quite well, and direct implementation yields a circuit that computes integer reciprocals in size $O(M(n)\log n)$ and depth $O(\log^2 n)$. The $\log n$ multiplier in the size comes from the fact that $\Theta(\log n)$ iterations of equation (1.5) are needed, each of which requires a multiplication of n bit values.

Noticing that the approximation y_i is very inaccurate in the early stages, it seems pointless to do calculations with all the erroneous bits of y_i. In fact, this observation produces a new algorithm which removes the $\log n$ multiplier from the size bound above; the algorithm that accomplishes this is shown in figure 1.1, and proofs of correctness and complexity are given in theorem 1.1. The approximation formula used in figure 1.1 looks different from that in equation (1.5), but the only difference is due to the new scaling required by having only $\frac{n}{2}$ bits for y_i.[2] The "for loop" in algorithm RECIP1 is also a new addition; it is present to overcome errors induced by using fixed point approximation to real numbers. The usefulness of this adjustment stage will become apparent from the proof of theorem 1.1.

[2]One way to view this is that now the precision of the fixed point representation is changed at each stage; at the smallest stage, the fractional precision is only $\pm\frac{1}{2}$, but at the next stage the precision is $\pm\frac{1}{4}$, and then $\pm\frac{1}{16}, \pm\frac{1}{256}, \ldots$

Algorithm RECIP1(x, n);
  if $n = 1$
    then begin
      $y \leftarrow 4$;
      end;
    else begin
      $t \leftarrow$ RECIP1$(\lfloor \frac{x}{2^{n/2}} \rfloor, \frac{n}{2})$;
      $y \leftarrow \left\lfloor \frac{2^{\frac{3}{2}n+1}t - xt^2}{2^n} \right\rfloor$;
      for $i \leftarrow 3$ **downto** 0 **do**
        if $(x(y + 2^i) \leq 2^{2n})$
          then begin
            $y \leftarrow y + 2^i$;
            end;
      end;
  return (y);
end.

FIGURE 1.1
Algorithm RECIP1.

For the remainder of this section, as well as in sections 1.4 and 1.5, the n-bit input x is assumed to satisfy $2^{n-1} \leq x < 2^n$ (i.e., the high-order bit is set). The algorithms may be modified so they do not require this assumption by simply shifting x (by bits) into the appropriate range, performing the algorithms found in this chapter, and shifting the results back into the proper range. The complexity of the shifting stages is negligible compared to the complexity of the algorithms discussed. Similarly, it is assumed that n is a power of 2.

THEOREM 1.1

Algorithm RECIP1 *in figure 1.1 correctly computes the integer reciprocal of x, and is realized with a circuit family of size $O(M(n))$ and depth $O(\log^2 n)$.*

PROOF

The following proof of the correctness is rather tedious; this comes from the fact that fixed point approximations to real numbers are used, so small errors (from rounding or truncating) are introduced at various points. A very simple way to get a feeling for why this method works is to examine how the error of an approximation is affected by equation (1.4); while this is not a proof that algorithm RECIP1 is correct, it does provide insight that is useful if the following proof is found to be confusing.

To simplify notation, let r represent the value returned by RECIP1(x, n). To prove the correctness of RECIP1, it is necessary to show that $r = \left\lfloor \frac{2^{2n}}{x} \right\rfloor$; in other words, $xr = 2^{2n} - s$ where $0 \leq s < x$. The proof is by induction on n; the correct value for $n = 1$ is stated explicitly in the algorithm.

Assume that the algorithm returns a correct value for inputs of size $\frac{n}{2}$. Let t be the value of RECIP1$(\left\lfloor \frac{x}{2^{n/2}} \right\rfloor, \frac{n}{2})$ as in figure 1.1, and let $d = 2^{\frac{3}{2}n+1}t - xt^2$. Also, denote the most significant $\frac{n}{2}$ bits by x_1 and the least significant $\frac{n}{2}$ bits by x_0, so $x = x_1 2^{n/2} + x_0$. The value d can now be written as

$$d = 2^{\frac{3}{2}n+1}t - t^2(x_1 2^{n/2} + x_0).$$

The value of interest in this proof is xr, so first we will find xd and then bound the difference between this and xr.

$$xd = 2^{2n+1}x_1 t + 2^{\frac{3}{2}n+1}x_0 t - t^2(x_1 2^{n/2} + x_0)^2$$

Using the induction hypothesis (that $x_1 t = 2^n - s'$, where $0 \leq s' < x_1$), this can be simplified to

$$xd = 2^{3n} - (2^{n/2}s' - tx_0)^2.$$

Dividing by 2^n, the result is

$$\frac{xd}{2^n} = 2^{2n} - (s' - \frac{tx_0}{2^{n/2}})^2.$$

Noting that s' and $\frac{tx_0}{2^{n/2}}$ are both positive and that the difference of these two is squared, it is possible to bound

$$\left(s' - \frac{tx_0}{2^{n/2}}\right)^2 \leq \max\left\{(s')^2, \left(\frac{tx_0}{2^{n/2}}\right)^2\right\}.$$

By the induction hypothesis, $s' < x_1 < 2^{n/2}$, so $(s')^2 < 2^{n/2}x_1 \leq x$. Furthermore,

$$\left(\frac{tx_0}{2^{n/2}}\right)^2 \leq \left(\frac{2^{n/2}x_0}{x_1}\right)^2 < \left(2^{n/2+1}\right)^2 = 2^{n+2} \leq 8x,$$

so $(s' - \frac{tx_0}{2^{n/2}})^2 < 8x$. In other words,

$$\frac{xd}{2^n} > 2^{2n} - 8x.$$

Now, considering the value y calculated by the Newton approximation equation,

$$xy = x\left\lfloor\frac{d}{2^n}\right\rfloor > x(\frac{d}{2^n} - 1) = \frac{xd}{2^n} - x > 2^{2n} - 9x$$

The adjustment stage of RECIP1 will adjust the least significant four bits of y to the correct value, as long as $\mathsf{RECIPROCAL}(x, n) - y \leq 15$ entering the adjustment stage. It has just been shown that, in fact, $\mathsf{RECIPROCAL}(x, n) - y \leq 9$, so RECIP1 correctly returns the integer reciprocal.

The complexity of the circuit is very straightforward to calculate. To calculate the size, notice that RECIP1 performs only a constant number of multiplications and simpler operations on $O(n)$ bit numbers in addition to the recursive call. In other words, the recurrence

$$S(n) \leq S(\frac{n}{2}) + cM(n) \tag{1.6}$$

$$S(1) = 1$$

describes the size of the circuit for RECIP1. The solution to equation (1.6) is given by

$$S(n) \leq c \sum_{i=0}^{\log n} M(\frac{n}{2^i}).$$

The earlier assumption that $M(n)$ satisfies equation (1.2) implies that $M(\frac{n}{2^i}) \leq \frac{1}{2^i} M(n)$, so the resulting size is $S(n) = O(M(n))$.

The depth of each level of recursion is bounded by $O(\log n)$, and since there are $\log n$ stages, the total depth is bounded by $O(\log^2 n)$. This is a rather simplistic depth analysis, but closer examination shows that this is the tightest upper bound possible. ∎

1.3 Integer Powering

The seemingly unrelated problems of integer reciprocal and integer powering are actually very closely related. In fact, it has been shown by Beame, Cook, and Hoover [3] that the two problems are equivalent with respect to constant depth reductions.[3] A survey of the research on integer division shows that all known efficient reciprocal algorithms use powering as an integral part (see the references for information on other reciprocal algorithms).

As an introduction to powering, consider a simple powering algorithm; the problem is to raise an n-bit number x to the m-th power, where $m \leq n$. Now write m in its binary notation, so $m = m_{\lfloor \log m \rfloor} 2^{\lfloor \log m \rfloor} + \cdots + m_2 2^2 + m_1 2^1 + m_0 2^0$. An algorithm (called repeated squaring) that takes advantage of the binary representation of m is shown in figure 1.2.

The complexity analysis of this algorithm is particularly easy, resulting in a circuit family with size $O(M(nm))$ and depth $O(\log n \log m)$. Note that this is considerably better than simply multiplying x by itself m times which takes size $O(mM(nm))$ and depth $\Theta(m \log n)$.

With this algorithm in mind, consider the reciprocal algorithm of the previous section; at first glance, RECIP1 doesn't seem to take any powers greater than squaring y_i. However, if a more global view is invoked, this squared term is again squared in the next stage, and repeatedly squared until

[3] A constant depth reduction is similar to the constant size and depth reduction mentioned earlier in this chapter, except that the size can increase by a polynomial amount.

```
Algorithm REPEATSQ(x, m);
    {Consider m in its binary representation:
                    m = m_{⌊log m⌋} 2^{⌊log m⌋} + ··· + m_2 2^2 + m_1 2^1 + m_0 2^0}
    i ← 0;
    p ← x;
    y ← 1;
    while i ≤ log n do begin
        if m_i = 1
          then begin
            y ← yp;
            end;
        p ← p^2;
    end;
end.
```

FIGURE 1.2
Repeated squaring method of taking powers.

the end of the algorithm. In other words, the algorithm actually takes large powers using the repeated squaring algorithm! An observant reader would have noticed that the depth of the algorithm RECIP1 is the same as the depth of the algorithm REPEATSQ (with $m = n$). Now it can be seen that this is no coincidence—RECIP1 was actually performing operations almost identical to REPEATSQ.

An interesting question now arises: Can reciprocals be computed in depth smaller than $\Omega(\log^2 n)$ if there were an algorithm for computing powers in small depth? Indeed, this is the case (more information on this will be presented in following sections); unfortunately, finding small depth circuits for powering seems to be as hard as looking at the reciprocal problem directly. What follows is a description of a powering algorithm that only requires $O(\log n \log\log n)$ depth (for $m = n$); the algorithm is rather confusing to people who haven't seen anything like it before. A good "warm-up" exercise would be to read and understand the multiplication algorithm of Schönhage and Strassen (see the references). The algorithm presented here consists of two parts: reducing the size of the input number x, and reducing the power.

1.3.1 Bit Reduction

Again, we wish to raise an n-bit number x to a power m, where $m \leq n$. The number x has at most $d = \left\lfloor \frac{n}{\log b} \right\rfloor + 1$ digits in base b notation and can be written as

$$x = x_{d-1}b^{d-1} + \cdots + x_2 b^2 + x_1 b + x_0. \tag{1.7}$$

If an indeterminate z is substituted for the occurrences of b that are raised to a power, then x can be represented by a polynomial $p(z) = x_{d-1}z^{d-1} + \cdots + x_2 z^2 + x_1 z^1 + x_0$, where $p(b) = x$.

Operations with such polynomials mirror the same operations performed on the numbers themselves, so for example, if x is represented by $p(z)$ and y is represented by $q(z)$, then the product of the two polynomials has the property that $p(z)q(z)|_{z=b} = p(b)q(b) = xy$.[4] The current interest is in powering, and it can be noticed that if x is represented by $p(z)$ and m is an integer, then $[p(z)]^m|_{z=b} = [p(b)]^m = x^m$. Efficient polynomial arithmetic is made possible by a domain change through Fourier transforms; we now see how this is done.

Returning to the original problem, let x be an n-bit number, where n is a power of 2—say $n = 2^p$. The input x can be broken into $k = 2^r$ blocks of $l = 2^{p-r}$ bits each, so letting $b = 2^l$, equation (1.7) becomes

$$x = x_{k-1}2^{l(k-1)} + \cdots + x_2 2^{2l} + x_1 2^l + x_0.$$

The polynomial representation of x (as described above) is therefore $p(z) = x_{k-1}z^{k-1} + \cdots + x_2 z^2 + x_1 z + x_0$; notice that $p(2^l) = x$.

To raise x to the mth power, simply find the polynomial $[p(z)]^m$ and evaluate at $z = 2^l$. Unfortunately, the polynomial $[p(z)]^m$ has degree $m(k-1)$, which is too large for an efficient powering algorithm (it is an interesting exercise to follow the development of the powering algorithm using all terms of $[p(z)]^m$ to see exactly where things go amiss).

Consider calculating $[p(z)]^m (\bmod z^k - 1)$. When the value $z = 2^l$ is inserted, the result is $x^m (\bmod 2^n - 1)$; by padding the input with zeros and increasing n to insure that $2^n - 1 > x^m$, this method produces the exact answer. Furthermore, polynomials modulo $z^k - 1$ never have degree greater than $k - 1$, so the problem of growing polynomial degrees has disappeared. With this in mind, the subject of most of the remainder of this section will be the problem of modular powering.

[4] The notation $p(x)|_{x=a}$ means the polynomial $p(x)$ evaluated at $x = a$; in other words, $p(a)$. Similarly, $p(z)q(z)|_{z=b}$ means to multiply the polynomials $p(z)$ and $q(z)$, and evaluate the resulting polynomial at $z = b$.

Let $b(z) = b_{m(k-1)}z^{m(k-1)} + \cdots + b_2z^2 + b_1z + b_0$ be the exact value of $[p(z)]^m$, and let $d(z) = d_{k-1}z^{k-1} + \cdots + d_1z + d_0$ be the reduction of $b(z)$ modulo $z^k - 1$ so that $d(z)$ has degree less than k. Since $z^k \equiv 1 (\bmod z^k - 1)$, it is easy to see that for $i = 0, 1, \ldots, k-1$,

$$d_i = \sum_{j=0}^{m-1} b_{jk+i}.$$

All b_i with $i > m(k-1)$ are assumed to be zero. Let $D = (d_0, d_1, \ldots, d_{k-1})$ and $X = (x_0, x_1, \ldots, x_{k-1})$ denote vectors of the coefficients of $d(z)$ and $p(z)$, respectively. The following lemma demonstrates an efficient way of computing the modular power polynomial $d(x)$ using Discrete Fourier Transforms (DFTs).[5]

LEMMA 1

Let $\mathsf{DFT}_k(X) = (t_0, t_1, \ldots, t_{k-1})$. *Then* $\mathsf{DFT}_k^{-1}((t_0^m, t_1^m, \ldots, t_{k-1}^m)) = D$.

PROOF

By the definition of the DFT,

$$t_i = \sum_{j=0}^{k-1} x_j\omega^{ij} = p(\omega^i)$$

for all $i = 0, 1, \ldots, k-1$, where ω is a principal kth root of unity. Raising each t_i to the mth power gives

$$t_i^m = [p(\omega^i)]^m = [p(z)]^m|_{z=\omega^i} = \sum_{j=0}^{m(k-1)} b_j\omega^{ij} = \sum_{p=0}^{m-1}\sum_{q=0}^{k-1} b_{pk+q}\omega^{i(pk+q)}.$$

But $\omega^{i(pk+q)} = (\omega^k)^{ip}\omega^{iq} = \omega^{iq}$ since ω is a kth root of unity, so

$$t_i^m = \sum_{p=0}^{m-1}\sum_{q=0}^{k-1} b_{pk+q}\omega^{iq} = \sum_{q=0}^{k-1}\left(\sum_{p=0}^{m-1} b_{pk+q}\right)\omega^{iq} = \sum_{q=0}^{k-1} d_q\omega^{iq}.$$

By the definition of the DFT, this is simply the ith term of $\mathsf{DFT}_k(D)$. As this holds for all $i = 0, 1, \ldots, k-1$, then $\mathsf{DFT}_k^{-1}((t_0^m, t_1^m, \ldots, t_{k-1}^m)) = D$.

■

[5] If the reader is unfamiliar with the Fourier transform or the Fast Fourier Transform algorithm, an introductory level discussion can be found in (Aho et al., [1]).

The Fourier transform of a k-vector (representing a degree $k-1$ polynomial) requires a principal kth root of unity ω. The polynomials that represent integers have integer coefficients, and to avoid doing computations over the complex field, it is possible to use finite rings as the basis of our computation. The ring of integers modulo 2^k-1 has a principal kth root of unity of $\omega = 2$, giving this ring the further nice property that multiplication by powers of ω is easily accomplished by bit shifts. Since computations on each element of X are now done modulo 2^k-1 it is clear how the original problem (powering an n-bit number modulo 2^n-1) is reduced to smaller subproblems (powering k-bit numbers modulo 2^k-1). This reduction can be repeated until the size of the subproblems is trivial. Furthermore, $k^{-1}(\bmod 2^k-1)$ exists by insuring that k is a power of 2, so the inverse DFT is possible.

A problem arises from the fact that the previous discussion of powering assumes that the *exact* values for the coefficients of $[p(z)]^m(\bmod z^k-1)$ are known, and the previous paragraph refers to only finding the coefficients modulo 2^k-1. The following lemma addresses this problem by showing how large to make k to insure that the coefficients are uniquely represented in this ring (i.e., the coefficients are less than 2^k-1).

LEMMA 2

The coefficients of $[p(z)]^m(\bmod z^k-1)$ *are less than* 2^k-1 *if*

$$2^r - r(m-1) - lm > 0 \tag{1.8}$$

(where r*,* l*, and* m *are defined in the preceding text).*

PROOF

First, it is proved by induction that the coefficients of $[p(z)]^m(\bmod z^k-1)$ are less than or equal to $k^{m-1}(2^l-1)^m$ for $m = 1, 2, \ldots$. The basis of the induction is easy; simply let $m = 1$ and the claimed bound becomes 2^l-1. The coefficients of $p(z)$ are all less than or equal to 2^l-1 since each coefficient is l bits long.

Now assume the claim is true for $m-1$ (that is, the coefficients of $[p(z)]^{m-1}(\bmod z^k-1)$ are less than or equal to $k^{m-2}(2^l-1)^{m-1}$). Let the expansion of $p(z)$ and $[p(z)]^{m-1}(\bmod z^k-1)$ be as follows:

$$\begin{aligned} p(z) &= x_{k-1}z^{k-1} + x_{k-2}z^{k-2} + \cdots + x_2z^2 + x_1z + x_0 \\ [p(z)]^{m-1} &= y_{k-1}z^{k-1} + y_{k-2}z^{k-2} + \cdots + y_2z^2 + y_1z + y_0 \end{aligned}$$

Notice that since $z^i \equiv z^{i(\bmod k)} (\bmod z^k - 1)$,

$$[p(z)]^m = [p(z)][p(z)]^{m-1} \equiv \sum_{i=0}^{k-1} \left(\sum_{j=0}^{k-1} x_j y_{i-j(\bmod k)} \right) z^i (\bmod z^k - 1).$$

Regardless of the particular values of i and j, it must be true that $x_j \leq 2^l - 1$ and $y_{i-j(\bmod k)} \leq k^{m-2}(2^l-1)^{m-1}$ (by the induction hypothesis), so $x_j y_{i-j(\bmod k)} \leq k^{m-2}(2^l-1)^m$. Since there are k terms like this added together for each coefficient of $[p(z)]^m (\bmod z^k - 1)$, each coefficient must be less than or equal to $k^{m-1}(2^l-1)^m$, and the proof by induction is finished.

Returning to the lemma, condition (1.8) states that $2^r > r(m-1) + lm$. In other words, taking each side as an exponent, $2^{(2^r)} > 2^{r(m-1)}2^{lm}$, and since $k = 2^r$ this implies that $k^{m-1}2^{lm} < 2^k$. Loosening the inequality slightly, this implies that (for $m \geq 1$)

$$k^{m-1}(2^l - 1)^m < 2^k - 1. \tag{1.9}$$

The previous inductive proof showed the coefficients of $[p(z)]^m (\bmod z^k - 1)$ must be less than or equal to the left hand side of inequality (1.9), so each coefficient must also be less than $2^k - 1$, completing the proof of the lemma. ∎

As an example of the reduction technique just described, consider a single stage of bit reduction as shown in figure 1.3. The value for k comes from calculations involving lemma 2; lemma 3 shows how this works. Notice the call on MODPOWER in REDUCE1—this is a recursive call that is left unspecified for the moment. As it turns out, a second type of reduction will be needed for efficient powering, and the recursive call (named MODPOWER here) may be on a *different* type of reduction. Notice the new assumption that $m \leq n^{\frac{3}{8}}$. This assumption simply makes the lemma easier to prove, and will not affect the final powering result at all (in fact, it will become apparent that this is the result of passing the assumption $m \leq n$ down through several lemmas).

LEMMA 3

Let $m \geq 16$ and $m \leq n^{\frac{3}{8}}$. Then assuming that MODPOWER(t_i, m, k) *correctly returns $t_i^m (\bmod 2^k - 1)$, the reduction* REDUCE1 *shown in figure 1.3 correctly returns $x^m (\bmod 2^n - 1)$. Furthermore, if the call on*

Algorithm REDUCE1(x, m, n);
    $p \leftarrow \log n$;
    $q \leftarrow \lceil \log m \rceil$;
    $r \leftarrow \lceil \frac{p}{2} + \frac{2q}{3} \rceil$;
    $k \leftarrow 2^r$;
    Divide x into k blocks of $l = 2^{p-r}$ bits each as $(x_0, x_1, \ldots, x_{k-1})$;
    $(t_0, t_1, \ldots, t_{k-1}) \leftarrow \mathsf{DFT}_k(x_0, x_1, \ldots, x_{k-1})$;
    for all $i = 0, 1, \ldots, k-1$ **pardo begin**
        $u_i \leftarrow$ MODPOWER(t_i, m, k);
    end;
    $(y_0, y_1, \ldots, y_{k-1}) \leftarrow \mathsf{DFT}_k^{-1}(u_0, u_1, \ldots, u_{k-1})$;
    $y \leftarrow y_0 + y_1 2^l + y_2 2^{2l} + \cdots + y_{k-1} 2^{(k-1)l} (\bmod 2^n - 1)$;
    return (y);
end.

FIGURE 1.3
Powering reduction style 1.

MODPOWER *requires size* $S(m,k)$ *and depth* $D(m,k)$*, then* REDUCE1 *requires total size* $kS(m,k) + O(nm^{\frac{4}{3}} \log n)$ *and total depth* $D(m,k) + O(\log n)$.

PROOF

The correctness of REDUCE1 follows directly from the previous discussion with the important points being lemma 1 and lemma 2. The only verification that needs to be done is that the condition (1.8) of lemma 2 holds; that is, that $2^r - r(m-1) - lm > 0$. What follows is basically an exercise in minimizing the function on the left hand side.

From figure 1.3, let $r = \lceil \frac{p}{2} + \frac{2q}{3} \rceil$. To avoid the ceiling function write r as $\frac{p}{2} + \frac{2q}{3} + \epsilon$, where ϵ is some value satisfying $0 \leq \epsilon < 1$. Obviously, if condition (1.8) holds for all ϵ in this interval, then the condition must also hold with the ceiling. Substituting this value for r and letting $m = 2^q$ and $l = 2^{p-r}$, the left hand side of condition (1.8) becomes

$$f(p,q,\epsilon) = 2^{\frac{p}{2} + \frac{2q}{3} + \epsilon} - \left(\frac{p}{2} + \frac{2q}{3} + \epsilon \right) (2^q - 1) - 2^{\frac{p}{2} + \frac{q}{3} - \epsilon}.$$

This formula is quite messy, but can be simplified greatly just by taking the partial derivative with respect to ϵ (which will reveal a lot of useful information).

$$\frac{\partial f}{\partial \epsilon}(p,q,\epsilon) = 2^{\frac{p}{2}+\frac{q}{3}} \ln 2 \left(2^{\epsilon+\frac{q}{3}} + 2^{-\epsilon}\right) - (2^q - 1)$$

This function is easily minimized for a given p and q (for an easy trick, substitute $t = 2^\epsilon$ and minimize with respect to t) when $\epsilon = -\frac{q}{6}$. Substituting this value into $\frac{\partial f}{\partial \epsilon}$, for any p and q the minimum value of the partial derivative with respect to ϵ is

$$2^{\frac{p}{2}+\frac{q}{2}+1} \ln 2 - (2^q - 1)\,. \tag{1.10}$$

In terms of p and q, the assumption that $m \leq n^{\frac{3}{8}}$ translates to $q \leq \frac{3p}{8}$ (or equivalently, that $p \geq \frac{8q}{3}$). We wish to show that equation (1.10) is greater than zero for all valid p and q. For any given q, equation (1.10) is minimum when p is at its minimum—in other words, when $p = \frac{8q}{3}$. Making this substitution, equation (1.10) becomes

$$2^{\frac{11q}{6}+1} \ln 2 - (2^q - 1),$$

which is easy to show greater than zero for all $q \geq 0$.

The past few paragraphs have shown that for all valid p, q, and ϵ, the derivative $\frac{\partial f}{\partial \epsilon}(p,q,\epsilon) > 0$. In other words, for all valid p and q, $f(p,q,\epsilon)$ is increasing in ϵ; therefore, for all valid p, q, and ϵ,

$$f(p,q,\epsilon) \geq f(p,q,0) = 2^{\frac{p}{2}+\frac{2q}{3}} - 2^{\frac{p}{2}+\frac{q}{3}} - \left(\frac{p}{2} + \frac{2q}{3}\right)(2^q - 1)\,.$$

Differentiating the right hand side with respect to p gives

$$2^{\frac{p}{2}} \frac{\ln 2}{2} \left(2^{\frac{2q}{3}} - 2^{\frac{q}{3}}\right) - \frac{2^q - 1}{2}.$$

This is obviously increasing in p, so is minimized when p is minimum; after making the substitution $p = \frac{8q}{3}$ and doing some rearranging, the above becomes

$$\frac{1}{2}\left[\left(2^{2q} - 2^{\frac{5q}{3}}\right) \ln 2 - (2^q - 1)\right],$$

which is easily shown to be greater than zero for all $q \geq 4$. In other words, for a given $q \geq 4$, $f(p,q,0)$ is increasing in p, so to minimize $f(p,q,0)$, again set p to $\frac{8q}{3}$. Therefore,

$$f(p,q,0) \geq f(\frac{8q}{3},q,0) = 2^{2q} - 2^{\frac{5q}{3}} - 2q\,(2^q - 1)\,,$$

which is greater than zero for all $q \geq 4$.

Summarizing, it has been shown that for all valid p, q, and ϵ,

$$f(p,q,\epsilon) \geq f(p,q,0) \geq f(\frac{8q}{3},q,0) \geq 0,$$

so condition (1.8) must hold, and REDUCE1 gives the correct answer by lemma 2, lemma 1, and the properties of polynomials discussed in the text before lemma 1.

The complexity of REDUCE1 relies on two results beyond the scope of this chapter: namely, the DFT_k and DFT_k^{-1} can be computed in size $O(k^2 \log k)$ and depth $O(\log n)$, and the evaluation of $[p(z)]^m|_{z=2^l}$ can be done in size $O(k^2)$ and depth $O(\log n)$. In other words, all steps except the call on MODPOWER can be done in size $O(k^2 \log k)$ and depth $O(\log n)$. Furthermore, since

$$k = 2^{\lceil \frac{p}{2}+\frac{2q}{3} \rceil} \leq 2^{\frac{p}{2}+\frac{2q}{3}+1} = 2n^{\frac{1}{2}}m^{\frac{2}{3}},$$

the above size can be written as $O(nm^{\frac{4}{3}} \log n)$. Including the size for the k calls on MODPOWER, the resulting size is $kS(m,k) + O(nm^{\frac{4}{3}} \log n)$. All recursive calls are done in parallel, so the total depth is $D(m,k) + O(\log n)$. ∎

The problem with repeatedly applying REDUCE1 is that the requirements of lemma 3 make reduction to a trivial problem size impossible (since n must be at least $m^{\frac{8}{3}}$); however, it is possible to reduce the power as well as the number of bits.

1.3.2 Power Reduction

Consider raising a number x to the mth power. If m is a perfect square with $w = \sqrt{m}$, it is easy to see that $x^m = (x^w)^w$; unfortunately, m is usually not a perfect square. To handle the more common case, let $v = \lfloor \sqrt{m} \rfloor$ and calculate $(x^v)^v$. Of course, this is not the desired answer, but notice that if $e = m - v^2$ is the error in the exponent of this approximation, e can be easily bounded by

$$e = m - v^2 \leq ((v+1)^2 - 1) - v^2 = 2v = 2\lfloor \sqrt{m} \rfloor.$$

Letting $e' = \lfloor \frac{e}{2} \rfloor$, $x^{e'}$ can be computed, squared, and multiplied by x (if e is odd) to achieve x^e. Notice that this computation of x^e can be done in parallel with the computation of $(x^v)^v$, so the original problem has been reduced to

```
Algorithm REDUCE2(x, m, n);
    p ← ⌊√m⌋;
    In Parallel do part1, part2
        part1: begin
                   t ← MODPOWER(x, p, n);
                   u ← MODPOWER(t, p, n);
                   end;
        part2: begin
                   e ← m − p^2;
                   e' ← ⌊e/2⌋;
                   v ← MODPOWER(x, e', n);
                   if (2e' = e)
                     then begin
                       w ← v^2 (mod 2^n − 1);
                       end;
                     else begin
                       w ← xv^2 (mod 2^n − 1);
                       end;
                   end;
    y ← uw (mod 2^n − 1);
    return (y);
end.
```

FIGURE 1.4
Powering reduction style 2.

3 smaller powerings (each of which raises a number to a power less than or equal to $\sqrt{m}$) and a constant number of multiplications. This reduction is called REDUCE2 and is shown in figure 1.4.

The correctness of REDUCE2 follows easily from the above discussion, and the complexity analysis is simple, so the following lemma is stated without proof.

LEMMA 4

Assuming MODPOWER(t, m, n) *correctly returns* $t^m \pmod{2^n - 1}$ *for all* t, m, *and* n, *the reduction* REDUCE2 *shown in figure 1.4 correctly returns*

$x^m \pmod{2^n - 1}$. Furthermore, if the call on MODPOWER(t, m, n) *requires size $S(m, n)$ and depth $D(m, n)$, then* REDUCE *requires total size $3S(\sqrt{m}, n) + O(M(n))$ and total depth $2D(\sqrt{m}, n) + O(\log n)$.*

Again, there is a problem with using just REDUCE2—while the correct answer is returned, the number of subproblems grows too rapidly, and the depth of the powering circuit using just REDUCE2 is $\Theta(\log n \log m)$. Fortunately, in the design of REDUCE1 and REDUCE2 there were some subtle adjustments made (such as the choice for r in REDUCE1) that allow the two reductions to work very well together. Combining the two reductions is addressed in the following section.

1.3.3 Putting the Pieces Together

The final modular power algorithm consists of an initial reduction using REDUCE2 followed by a test to see if the power has been reduced to smaller than 16. If the power is less than 16, then the result can be computed using the REPEATSQ algorithm presented at the beginning of this section (taking size $O(M(n))$ and depth $O(\log n)$); otherwise, the subproblems are further reduced by two applications of REDUCE1. All three of these reductions can be viewed together as a single "composite reduction" that produces subproblems with reduced size (i.e., number of bits) and reduced power. A proof of the correctness of this algorithm, along with the complexity analysis, is given in the following theorem.

THEOREM 1.2

Let x be an n-bit integer, and m be an integer with $m^2 \leq n$. The algorithm just described computes $x^m \pmod{2^n - 1}$ in $O(nm^4 \log n \log\log n)$ size and $O(\log n + \log m \log\log m)$ depth.

PROOF

The correctness of the above algorithm is proved by induction on the number of complete composite reductions required before the power is reduced below 16. If no reductions are required, the result is correct by the correctness of algorithm REPEATSQ. Assume that $R \geq 1$ reductions are required—by the condition of the theorem, $m \leq n^{\frac{1}{2}}$, so after the first reduction using REDUCE2, each subproblem of raising an n-bit number to the m'th power is such that $m' \leq n^{\frac{1}{4}}$. (Note that this means the condition for lemma 3 is satisfied.)

After the first reduction via REDUCE1, each resulting subproblem has $k \geq n^{\frac{1}{2}}(m')^{\frac{2}{3}}$ bits. (Notice that

$$m' = (m')^{\frac{3}{4}}(m')^{\frac{1}{4}} \leq n^{\frac{3}{16}}(m')^{\frac{1}{4}} = \left(n^{\frac{1}{2}}(m')^{\frac{2}{3}}\right)^{\frac{3}{8}} \leq k^{\frac{3}{8}},$$

so the condition for lemma 3 is again satisfied.)

Following the second reduction via REDUCE1, each subproblem has $k' \geq k^{\frac{1}{2}}(m')^{\frac{2}{3}}$ bits; using the previous bounds for k, $(m')^2$ can be bounded as

$$(m')^2 = (m')(m') \leq n^{\frac{1}{4}}(m') = \left(n^{\frac{1}{2}}(m')^{\frac{2}{3}}\right)^{\frac{1}{2}}(m')^{\frac{2}{3}} \leq k^{\frac{1}{2}}(m')^{\frac{2}{3}} \leq k'.$$

In other words, after one composite reduction each subproblem of raising a k'-bit number to the m'th power satisfies $(m')^2 \leq k'$. Only $R-1$ composite reductions are required for these subproblems (since R reductions were required for the original problem), and since $(m')^2 \leq k'$, the induction hypothesis applies to say that all these subproblems are correctly solved.

Going backwards through each individual reduction in the composite reduction, it has been noted that the conditions for lemmas 3 and 4 have been satisfied, so the correctness of the algorithm follows directly from these lemmas.

Now examine the size required for this algorithm. Let $S(m,n)$ denote the size of raising an n-bit number to the mth power modulo $2^n - 1$. The result of applying the size of REDUCE2 (from lemma 4) to the size of REDUCE1 (from lemma 3) which is again applied to itself gives the size for one composite reduction. The result is (using k, k', and m' as defined above)

$$\begin{aligned} S(m,n) = 3kk'S(m',k') &+ O(k^2(m')^{\frac{4}{3}}\log k) \\ &+ O(n(m')^{\frac{4}{3}}\log n) + O(M(n)). \end{aligned}$$

Using the bounds $k \leq 2n^{\frac{1}{2}}(m')^{\frac{2}{3}}$ (see the proof of lemma 3) and $m' \leq m^{\frac{1}{2}}$, in addition to the new bound $k' \leq 2k^{\frac{1}{2}}(m')^{\frac{2}{3}} = 2^{\frac{3}{2}}n^{\frac{1}{4}}m'$, gives a size of

$$S(m,n) = 3kk'S(m',k') + O(nm^{\frac{4}{3}}\log n) + O(nm^{\frac{2}{3}}\log n) + O(M(n)).$$

Using the Schönhage and Strassen algorithm, we know that $M(n) = O(n\log n\log\log n)$, so this can be simplified greatly to

$$S(m,n) = 3kk'S(m',k') + O(nm^{\frac{4}{3}}\log n\log\log n).$$

Removing the big-O notation, the above size bound can be expressed (for some constant c) as

$$S(m,n) \leq 3kk'S(m',k') + cnm^{\frac{4}{3}} \log n \log\log n.$$

Notice that this size only applies if a complete composite reduction is performed (i.e., $m' \geq 16$ or $m \geq 256$). For $m < 256$, only a constant number of multiplications are required, so $S(m,n) = O(M(n))$.

The claim is that $S(m,n) \leq c'nm^4 \log n \log\log n$ for some c', and is proved by induction on m. For $m < 256$ and the appropriate c' and c'',

$$S(m,n) \leq c''M(n) \leq c'nm^4 \log n \log\log n,$$

so this serves as a basis for the induction. Now assume $m \geq 256$, and the induction hypothesis states that

$$S(m',k') \leq c'k'(m')^4 \log k' \log\log k'$$

for $m' < m$. Using the bound $k' \leq 2^{\frac{3}{2}} n^{\frac{1}{4}} m^{\frac{1}{2}}$ and noticing that $2^{\frac{3}{2}} m^{\frac{1}{2}} \leq m^{\frac{11}{16}}$ for $m \geq 256$, k' can now be bounded as $k' \leq n^{\frac{1}{4}} m^{\frac{11}{16}} \leq n^{\frac{19}{32}}$. This means that $\log k' \leq \frac{19}{32} \log n$, so using all the upper bounds,

$$\begin{aligned} 3(kk')S(m',k') &\leq 3\left(2^{\frac{5}{2}} n^{\frac{3}{4}} m^{\frac{5}{6}}\right)\left(c'2^{\frac{3}{2}} n^{\frac{1}{4}} m^{\frac{1}{2}} m^2 \frac{19}{32} \log n \log\log n\right) \\ &= \frac{57}{2} c'nm^{\frac{10}{3}} \log n \log\log n, \end{aligned}$$

so

$$\begin{aligned} S(m,n) &\leq \frac{57}{2} c'nm^{\frac{10}{3}} \log n \log\log n + cnm^{\frac{4}{3}} \log n \log\log n \\ &\leq \left(\frac{57}{2} c'm^{-\frac{2}{3}} + cm^{-\frac{8}{3}}\right) nm^4 \log n \log\log n. \end{aligned}$$

Since $m \geq 256$, this can be loosely upper bounded by

$$S(m,n) \leq \left(\frac{3}{4}c' + c\right) nm^4 \log n \log\log n,$$

and for $c' \geq 4c$ this becomes

$$S(m,n) \leq c'nm^4 \log n \log\log n,$$

proving the claimed size bound.

Turning to the depth, let $D(m,n)$ represent the depth of raising an n-bit number to the mth power modulo 2^n-1, and the depth of a composite reduction can be expressed as

$$D(m,n) = 2D(m',k') + O(\log n)$$

for $m \geq 256$ (i.e., $m' \geq 16$), and $D(m,n) = O(\log n)$ for $m < 256$. A depth bound of $D(m,n) \leq c'(\log n + \log m \log\log m)$ can be proved by induction; the basis follows easily for $m < 256$.

For $m \geq 256$, the induction hypothesis states that

$$D(m',k') \leq c'(\log k' + \log m' \log\log m').$$

Since $m' \leq m^{\frac{1}{2}}$, we can bound $\log m' \log\log m' \leq \frac{1}{2}\log m(\log\log m - 1)$, so

$$\begin{aligned} D(m',k') &\leq c'(\frac{3}{2} + \frac{1}{4}\log n + \frac{1}{2}\log m + \frac{1}{2}\log m(\log\log m - 1) \\ &= c'(\frac{3}{2} + \frac{1}{4}\log n + \frac{1}{2}\log m \log\log m). \end{aligned}$$

In other words, for some constant c,

$$\begin{aligned} D(m,n) &\leq 2D(m',k') + c\log n \\ &\leq \left(\frac{3c'}{\log n} + \frac{c'}{2} + c\right)\log n + c'\log m \log\log m. \end{aligned}$$

Since $\frac{3c'}{\log n} \leq \frac{3c'}{2\log m} \leq \frac{3c'}{16}$ for $m \geq 256$,

$$D(m,n) \leq \left(\frac{11}{16}c' + c\right)\log n + c'\log m \log\log m.$$

For $c' \geq \frac{16}{5}c$, this can be simplified to

$$D(m,n) \leq c'(\log n + \log m \log\log m),$$

proving the claimed depth bound. ∎

Returning to the original (exact) powering problem, the following easy corollary completes the study of integer powering.

COROLLARY 1

If x is an n-bit integer and m is an integer satisfying $m \leq n$, then x^m can be computed by a circuit of size $O(nm^5 \log n \log\log n)$ and depth $O(\log n + \log m \log\log m)$.

PROOF

Let $N = nm$. Since $m \leq n$, multiplying both sides of the inequality by m shows that $m^2 \leq nm = N$. By theorem 1.2, after padding x with zeros in the most significant $n(m-1)$ places, $x^m (\bmod 2^N - 1)$ can be computed in size $O(Nm^4 \log N \log\log N) = O(nm^5 \log n \log\log n)$ and depth $O(\log N + \log m \log\log m) = O(\log n + \log m \log\log m)$. Since x^m must be less than $2^{nm} - 1$, the modular computation actually gives the exact value of x^m. ∎

1.4
High Order Convergence with Newton Approximation

Given that repeated application of the Newton approximation formula given in Section 1.2 computes powers in a depth-inefficient way, it is worthwhile to examine how efficient powering methods can be incorporated to reduce the complexity of finding reciprocals.

Recall the approximation formula for finding *real* reciprocals given in equation (1.4). The initial ideas here are presented in terms of real reciprocals, and then the simple changes to the integer reciprocal problem are examined. Some algebraic manipulation shows that applying the approximation formula twice, the approximation refinement becomes

$$y_{i+2} = y_i(1 + (1 - xy_i) + (1 - xy_i)^2 + (1 - xy_i)^3).$$

In fact, the original equation can be rewritten as

$$y_{i+1} = y_i(1 + (1 - xy_i)),$$

with the basic pattern emerging of

$$y_{i+m} = y_i \sum_{j=0}^{2^m - 1} (1 - xy_i)^j. \tag{1.11}$$

(Of course, we haven't *proven* that this is the general form of repeated application of equation (1.4)—this is left to the interested reader. A proof that this equation, after scaling, gives the correct answer will be given in theorem 5.)

A nice property of equation (1.11) is that the upper limit of the sum does not necessarily have to be of the form $2^m - 1$ in order to work correctly.

We wish to view an application of equation (1.11) as a single approximation step, so the new approximation formula can be written as

$$y_{i+1} = y_i \sum_{j=0}^{k-1} (1 - xy_i)^j. \tag{1.12}$$

This equation is called the kth order Newton approximation formula; the name comes from the fact that convergence is of order k. Desirable convergence properties can be proven for equation (1.12), but as we are interested in integer reciprocals, the scaled version should be examined first. Performing fixed point scaling exactly as was done for the second order formula of Section 1.2 gives a fixed-point equation; however, as before, only a small number of bits of y_i need to be considered in the calculation of y_{i+1}. If we let $y_i = \mathsf{RECIPROCAL}(\lfloor \frac{x}{2^{n-d}} \rfloor, d)$ (i.e., the integer reciprocal of the d most significant bits of x), and $x' = \lfloor \frac{x}{2^{n-dk}} \rfloor$ (the dk most significant bits of x) then the resulting equation is

$$y_{i+1} = \left\lfloor \frac{y_i \sum_{j=0}^{2k-1} 2^{d(k+1)(2k-j-1)} (2^{d(k+1)} - x'y_i)^j}{2^{2dk^2}} \right\rfloor \tag{1.13}$$

Notice that here the upper limit on the sum is $2k - 1$ instead of $k - 1$—the upper limit has been raised to overcome the same type of problem that required the adjustment stage of RECIP1; however, equation (1.13) is still referred to as the kth order Newton approximation formula.

To construct an algorithm using equation (1.13), the exact order of each approximation step must be considered; this schedule of approximations depends on complexity considerations and will be addressed in the next section. The following lemma shows how equation (1.13) affects an approximation.

LEMMA 5

If $d \geq 2$ and $y_i = \mathsf{RECIPROCAL}(\lfloor \frac{x}{2^{n-d}} \rfloor, d)$, then equation (1.13) gives y_{i+1} that satisfies

$$0 \leq \mathsf{RECIPROCAL}(\left\lfloor \frac{x}{2^{n-dk}} \right\rfloor, dk) - y_{i+1} \leq 2.$$

Furthermore, equation (1.13) can be evaluated by a circuit family with size $O(dk^7 \log dk \log\log dk)$ and depth $O(\log dk + \log k \log\log k)$.

PROOF

This proof closely parallels the proof of theorem 1.1. Writing x' in two parts as $x' = x_1 2^{d(k-1)} + x_0$, the assumption on y_i states that $y_i = \mathsf{RECIPROCAL}(x_1, d)$, or that $x_1 y_i = 2^{2d} - s$, where $0 \le s < x_1$. This implies that $x'y_i = (x_1 2^{d(k-1)} + x_0)y_i = 2^{d(k+1)} - (2^{d(k-1)}s - x_0 y_i)$. To simplify notation, let $w = 2^{d(k+1)}$ and $z = (2^{d(k-1)}s - x_0 y_i)$, so $x'y_i = w - z$.

Let

$$d = y_i \sum_{j=0}^{2k-1} 2^{d(k+1)(2k-j-1)} (2^{d(k+1)} - x'y_i)^j = y_i \sum_{j=0}^{2k-1} w^{2k-j-1} z^j.$$

The quantity of interest is $x'y_{i+1}$, so first compute $x'd$ as

$$\begin{aligned} x'd &= (w-z) \sum_{j=0}^{2k-1} w^{2k-j-1} z^j = \sum_{j=0}^{2k-1} w^{2k-j} z^j - \sum_{j=0}^{2k-1} w^{2k-j-1} z^{j+1} \\ &= w^{2k} - z^{2k} = 2^{2dk(k+1)} - (2^{d(k-1)}s - x_0 y_i)^{2k}. \end{aligned}$$

Dividing by 2^{2dk^2} gives

$$\frac{x'd}{2^{2dk^2}} = 2^{2dk} - \left[\frac{s}{2^d} - \frac{x_0 y_i}{2^{dk}}\right]^{2k}.$$

Since $\frac{s}{2^d}$ and $\frac{x_0 y_i}{2^{dk}}$ are both positive, we can bound

$$\left|\frac{s}{2^d} - \frac{x_0 y_i}{2^{dk}}\right| \le \max\left\{\frac{s}{2^d}, \frac{x_0 y_i}{2^{dk}}\right\}. \tag{1.14}$$

The first of these terms is easy to bound: $\frac{s}{2^d} < 1$ since $s < x_1 < 2^d$. To bound the second term, notice that $y_i = \left\lfloor \frac{2^{2d}}{x_1} \right\rfloor \le \frac{2^{2d}}{x_1}$, so

$$\frac{x_0 y_i}{2^{dk}} \le \frac{x_0}{2^{d(k-2)} x_1} < \frac{2^{d(k-1)}}{2^{d(k-2)} 2^{d-1}} = 2.$$

Therefore, using equation (1.14),

$$\left(\frac{s}{2^d} - \frac{x_0 y_i}{2^{dk}}\right)^{2k} = \left(\left|\frac{s}{2^d} - \frac{x_0 y_i}{2^{dk}}\right|\right)^{2k} < 2^{2k}. \tag{1.15}$$

Since $d \ge 2$, this can be further bounded as

$$2^{2k} \le 2^{dk} < 2 \cdot 2^{dk-1} \le 2x'.$$

Notice that since the power $2k$ on the left hand side of equation (1.15) is even, the error term in equation (1.15) must be positive; in other words, $\frac{x'd}{2^{2dk^2}} \leq 2^{2dk}$. It follows that $2^{2dk} - 2x' < \frac{x'd}{2^{2dk^2}} \leq 2^{2dk}$.

The formula in equation (1.13) actually uses $\left\lfloor \frac{d}{2^{2dk^2}} \right\rfloor$, so

$$x'y_{i+1} = x' \left\lfloor \frac{d}{2^{2dk^2}} \right\rfloor > x' \left(\frac{d}{2^{2dk^2}} - 1 \right) = \frac{x'd}{2^{2dk^2}} - x' > 2^{2dk} - 3x'$$

If $\mathsf{RECIPROCAL}(x', dk) - y_{i+1} \geq 3$, then $x'y_{i+1} \leq 2^{2dk} - 3x'$. As just shown, this is impossible, so $\mathsf{RECIPROCAL}(x', dk) - y_{i+1} \leq 2$.

To evaluate equation (1.13), a circuit has to compute the jth power of $d(k+1)$ bit numbers, for $0 \leq j < 2k$. Noticing that for each j the size of this powering is $O(dkj^5 \log dk \log\log dk)$ from corollary 1, the total size required to take all the powers necessary is asymptotically upper-bounded by

$$\begin{aligned} \sum_{j=0}^{2k-1} cdkj^5 \log dk \log\log dk &= cdk \log dk \log\log dk \sum_{j=0}^{2k-1} j^5 \\ &< cdk^7 \log dk \log\log dk. \end{aligned}$$

As the reader can easily verify, the cost of adding these powers, multiplying by y_i, and scaling back down are all negligible compared the cost of powering, so the size of the circuit to evaluate equation (1.13) is $O(dk^7 \log dk \log\log dk)$.

All the powers are done in parallel, each having depth at most $O(\log dk + \log k \log\log k)$, and every other operation (the large sum and the rescaling) in the evaluation of equation (1.13) can be shown to have depth $O(\log dk)$; therefore, the total depth of evaluating equation (1.13) is $O(\log dk + \log k \log\log k)$. ∎

1.5 An Efficient Parallel Reciprocal Circuit

The results of the previous section can be used to design a parallel algorithm for finding reciprocals in depth $O(\log n \log\log n)$. In essence, lemma 5 says that an approximation to the reciprocal that is accurate to d bits can be extended to an accuracy of dk bits in $O(dk^7 \log dk \log\log dk)$ size and $O(\log dk + \log k \log\log k)$ depth.

To design a reciprocal algorithm, we need to come up with a sequence of approximation accuracies $d_1, d_2, d_3, \ldots$ such that after doing i approximation refinements, the result is accurate to d_i bits; eventually, all n bits should be known. In searching for criteria to design such a sequence, a desirable feature of parallel algorithms is that the work is spread out evenly across time. Looking at the form of the size bound from lemma 5, a good candidate is to set the size of each stage to $O(n \log n \log\log n)$. Setting $d_1 = 2$ (so two bit are known initially), the schedule then works out as

$$\begin{aligned} & d_i k_i^7 \log d_i k_i \ \log\log d_i k_i \ \leq n \log n \ \log\log n \\ \Longrightarrow\ & d_i \left(\frac{d_{i+1}}{d_i}\right)^7 \log d_{i+1} \ \log\log d_{i+1} \leq n \log n \ \log\log n \\ \Longrightarrow\ & d_{i+1}^7 \log d_{i+1} \ \log\log d_{i+1} \leq n d_i^6 \log n \ \log\log n \end{aligned}$$

Noticing that $d_{i+1} \leq n$ at all times (otherwise, the whole answer would be known!), the above inequality is satisfied with

$$d_{i+1} = n^{\frac{1}{7}} d_i^{\frac{6}{7}}.$$

Solving this recurrence (with the initial condition $d_1 = 2$) reveals that the sequence of accuracies is

$$d_i = 2n^{1-\left(\frac{6}{7}\right)^{i-1}}.$$

Unfortunately, this schedule does not produce just integers for accuracies (in fact, not necessarily even rational numbers!), so instead, let $m = \log n$ (recall that n is a power of 2 by assumption) and define the function

$$f(i) = \left\lfloor m \left(1 - \left(\frac{6}{7}\right)^{i-1}\right) \right\rfloor. \tag{1.16}$$

Then the schedule can be defined by

$$d_i = 2^{f(i)}. \tag{1.17}$$

The result is the algorithm shown in figure 1.5.

LEMMA 6

Algorithm RECIP2 *shown in figure 1.5 correctly computes the reciprocal of an n-bit number, and can be realized with a circuit family of size $O(n \log n (\log\log n)^2)$ and depth $O(\log n \ \log\log n)$.*

Algorithm RECIP2(x, n);
$m \leftarrow \log n$;
$d_1 \leftarrow 2$;
$i \leftarrow 2$;
if $(x \geq 3 \cdot 2^{n-2})$
then begin
$y_1 \leftarrow 5$;
end;
else begin
$y_1 \leftarrow 8$;
end;
while $i \leq \left\lceil \frac{\log\log n}{\log \frac{7}{6}} \right\rceil$ **do begin**
$t \leftarrow \left\lfloor m\left(1 - \left(\frac{6}{7}\right)^{i-1}\right)\right\rfloor$;
$d_i \leftarrow 2^t$;
$k_i \leftarrow \frac{d_i}{d_{i-1}}$;
$x' \leftarrow \left\lfloor \frac{x}{2^{n-d_i}} \right\rfloor$;
$y_{i+1} \leftarrow \left\lfloor \frac{y_i \sum_{j=0}^{2k_i-1} 2^{d_{i-1}(k_i+1)(k_i-j-1)}(2^{d_{i-1}(k_i+1)} - x'y_i)^j}{2^{2d_{i-1}k_i^2}} \right\rfloor$;
for $j \leftarrow 1$ **downto** 0 **do**
if $(x'(y_{i+1} + 2^j)) \leq 2^{2d_i}$
then begin
$y_{i+1} \leftarrow y_{i+1} + 2^j$;
end;
$i \leftarrow i + 1$;
end;
return (y_i);
end.

FIGURE 1.5
Algorithm RECIP2.

PROOF

The fact that algorithm RECIP2 meets the schedule of equation (1.17)

is a very simple proof by induction. The basis of the induction is trivial—the integer reciprocals of the two possible two-bit numbers are hard-wired into the algorithm. The induction step is proved by lemma 5 (notice the adjustment step in figure 1.5 that takes up the slack in possible error from lemma 5). The final answer after $p = \left\lceil \frac{\log\log n}{\log \frac{7}{6}} \right\rceil + 1$ steps is d_p bits. Computing $f(p)$ (where f is defined in equation (1.16)) shows that $f(p) = m$; in other words, $d_p = 2^m = n$.

By lemma 5, the size of stage i is $O(d_{i-1}k_i^7 \log d_{i-1}k_i \log\log d_{i-1}k_i)$. Examining k_i, the order of approximation at stage i is $k_i = 2^{f(i)-f(i-1)}$. Focusing on the exponent,

$$\begin{aligned} f(i) - f(i-1) &= \left\lceil m\left(\frac{6}{7}\right)^{i-2} \right\rceil - \left\lceil m\left(\frac{6}{7}\right)^{i-1} \right\rceil \\ &\le \left\lceil m\left(\frac{6}{7}\right)^{i-2} \right\rceil - \frac{6}{7}\left\lceil m\left(\frac{6}{7}\right)^{i-2} \right\rceil + 1 \\ &= \frac{1}{7}\left\lceil m\left(\frac{6}{7}\right)^{i-2} \right\rceil + 1. \end{aligned}$$

This means that

$$d_{i-1}k_i^7 \le 2^{m - \left\lceil m\left(\frac{6}{7}\right)^{i-2}\right\rceil + \left\lceil m\left(\frac{6}{7}\right)^{i-2}\right\rceil + 7} = O(n).$$

Furthermore, since $d_{i-1}k_i < n$, the total size of stage i (regardless of i) is $O(n \log n \log\log n)$. Over all $O(\log\log n)$ stages, the total size becomes $O(n \log n (\log\log n)^2)$.

By lemma 5, the depth of stage i is $O(\log d_{i-1}k_i + \log k_i \log\log k_i)$. Examining each term separately, the first term is $O(\log n)$ for all i, which produces a total depth of $O(\log n \log\log n)$ over all stages. In the second term, $\log\log k_i$ can be bounded by $\log\log n$ to obtain a depth over all stages of

$$\begin{aligned} \sum_{i=2}^{p} \log k_i \log\log k_i &\le \log\log n \sum_{i=2}^{p} \log k_i \\ &= \log\log n \sum_{i=2}^{p} [f(i) - f(i-1)] \\ &= \log\log n \,[f(p) - f(1)] = O(\log n \log\log n). \end{aligned}$$

Combining both terms of the depth, the total depth can be seen to be $O(\log n \log\log n)$. ∎

The algorithm RECIP2 just described is certainly an efficient reciprocal algorithm (in terms of both size and depth), but it does not clearly specify a relationship between the complexity of multiplication and that of division. (The similarity of the size bound with the size of the Schönhage-Strassen multiplication algorithm is mere coincidence.) In this sense, algorithm RECIP1 was better, since the size was closely tied to the size of multiplication (in fact, the size was $O(M(n))$). Can the good qualities of both algorithms (the size bound of RECIP1 and the small depth of RECIP2) be combined? Fortunately, the answer to this question is yes.

The new algorithm is RECIP3 shown in figure 1.6; the value N is the number of bits of the *original* problem (before any reductions). The basic idea behind algorithm RECIP3 is to use RECIP2 to find a sufficiently accurate initial estimate of the integer reciprocal so that only $O(\log\log n)$ stages of second order approximations are needed.

THEOREM 1.3

Algorithm RECIP3 *in figure 1.6 correctly computes the reciprocal of an n-bit number, and can be realized with a circuit family of size $O(M(n))$ and depth $O(\log n \log\log n)$.*

PROOF

Algorithm RECIP3 is a hybrid of RECIP1 and RECIP2, and the correctness follows directly from the correctness of those algorithms (see theorem 1.1 and lemma 6).

After i steps of recursion in RECIP3, $n = \frac{N}{2^i}$, so it only takes $\log(\log^2 N) = O(\log\log N)$ steps of second order reduction before $n \leq \frac{N}{\log^2 N}$. The complexity analysis of the second order stages is identical to theorem 1.1, but with only $O(\log\log N)$ stages. In other words, the size of the second order approximations (not counting the call on RECIP2) is $O(M(N))$ and the depth is $O(\log N \log\log N)$.

The size of the call on RECIP2 is easily computed from lemma 6 to be

$$O(\frac{N}{\log^2 N} \log \frac{N}{\log^2 N} \log\log \frac{N}{\log^2 N}) = O(N),$$

and the depth is $O(\log N \log\log N)$.

```
Algorithm RECIP3(x, n);
    if n ≤ N/log² N                          {N is the size of the original problem.}
      then begin
        y ← RECIP2(x, n);
        end;
      else begin
        t ← RECIP3(⌊x/2^{n/2}⌋, n/2);
        y ← ⌊(2^{(3/2)n+1} t − x t²)/2^n⌋;
        if (x(y + 1) ≤ 2^{2n})
          then begin
            y ← y + 1;
            end;
        end;
    return (y);
end.
```

FIGURE 1.6
Algorithm RECIP3.

Combining the complexity of the second order stages with the complexity of the call on RECIP2, the final result is that the circuit for RECIP3 has size $O(M(N))$ and depth $O(\log N \log\log N)$. ∎

1.6 Summary

This chapter examined the most complex of the basic arithmetic problems—division. The algorithm presented in this chapter is essentially that of Reif and Tate [14]; some minor changes have been made to clarify the presentation. While it can be shown that division is at least as hard as the other arithmetic problems (addition, subtraction, and multiplication), it is unknown whether division is strictly *harder* than the other operations. In comparison with multiplication (the second hardest problem), the results of theorem 1.3 show that while it may still be possible that division is harder than multiplication, the difference is not all that great (in terms of asymptotic growth).

There is potential for future research on division, either in finding a lower bound or in finding a better upper bound that could possibly match that of multiplication (as is the case sequentially).

For further information, the interested reader can consult Pippenger [11] for an excellent summary of computing arithmetic functions with various circuit models. For more in-depth treatment of multiplication, consult Schönhage and Strassen [15]. Sequential treatment of the basic arithmetic problems (including Schönhage and Strassen's multiplication algorithm), as well as an introduction to the Fast Fourier Transform, is covered in Aho et al. [1]. For a historical tour through the development of division algorithms (most of which use variations on the method described in Exercise 1.6), consult Cook [7], Reif [12], Beame et al. [3], Reif [13], Hastad and Leighton [9] Melhorn and Preparata [10], and Shankar and Ramachandran [16]. Treatment of more complex functions (such as square root and logarithms) can be found in Alt [2]. For various approaches to the polynomial reciprocal problem, see Bini and Pan [4] and Eberly [8] as well as Reif and Tate [14] which describes the algorithm hinted at by Exercise 1.7. Recently a new, highly efficient, $O(\log n \log^* n)$ time algorithm for finding polynomial reciprocals has been found by Bini and Pan [5]. Finally, the efficient powering circuit described in Exercise 1.8 is due to Hui Chen [6].

1.7 Exercises

1.1 Prove that the integer division problem can be solved by finding a single integer reciprocal and performing a constant number of multiplications and divisions.

1.2 Derive an equation that describes how the error is affected by

a) equation (1.4)
b) equation (1.12).

What assumptions must be made on the value of x in order for the iteration equations to converge to the appropriate answer?

1.3 The depth of any bounded fan-in circuit family for multiplication is $\Omega(\log n)$. Calculate the depth of algorithm RECIP1 more exactly than was done in theorem 1.1 to show that it is $\Omega(\log^2 n)$.

1.4 Since two reductions by REDUCE1 give subproblems with approximately $n^{1/4}m$ bits, why couldn't we define r in REDUCE1 at $\frac{1}{4}p + q$ so as to only require *one* reduction? How does using two reductions overcome this problem (what is the number of subproblems produced in each case)?

1.5 How would the error estimate of lemma 5 be affected if the upper limit on the sum in equation (1.13) was $k - 1$ instead of $2k - 1$?

1.6 There are, of course, other ways of computing reciprocals than with the algorithm presented in this chapter. The purpose of this exercise is to derive a different algorithm for computing reciprocals. For simplicity, assume we are solving the real reciprocal problem using fixed point binary representation. The input x takes n bits to represent and is assumed to be in the range $(0, 1)$. We want the answer z such that $|z - \frac{1}{x}| < 2^{-n}$.

a) Derive the Maclaurin series expansion for $f(u) = \frac{1}{1-u}$. For what values of u does this series converge?
b) Use this series to design an algorithm that computes the reciprocal of x. The range of values for u should be restricted so that only $O(n)$ terms of the series need to be computed to achieve the desired accuracy.
c) Using the powering complexity from corollary 1, what is the complexity of this reciprocal algorithm?

1.7 Let $R = \{D, +, \cdot, 0, 1\}$ be a ring, and let $p(x)$ be a degree n polynomial in $R[x]$. The polynomial reciprocal problem on input $p(x)$ is defined as computing the unique polynomial $q(x)$ such that

$$x^{2n-2} = q(x)p(x) + r(x),$$

where the degree of $r(x)$ is less than n. The floor notation can be used to equivalently state

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

The model of computation for this problem is a circuit where each node can perform addition, multiplication, or reciprocation (when it is defined) in the ring R.

A form of Newton iteration can be used to solve this problem, and provides good sequential results. Consider the polynomial $p(x)$ in "halves", so $p(x) = p_1(x)x^{n/2} + p_0(x)$, and let $q_1(x) = \mathsf{PRECIP}(p_1(x))$. Then the reciprocal of $p(x)$ can be calculated by

$$\mathsf{PRECIP}(p(x)) = \left\lfloor \frac{2q_1(x)x^{(3/2)n-2} - p_0(x)(q_1(x))^2}{x^{k-2}} \right\rfloor .$$

Notice how similar this formula is to the second-order integer iteration formula of algorithm RECIP1. Decide what is meant by a "high-order" formula for this problem, and derive the iteration formula. Use this formula to design an efficient parallel algorithm for the polynomial reciprocal problem.

In analyzing the complexity of this algorithm, let $PM(n)$ be the size complexity of multiplying two degree n polynomials in depth $O(\log n)$. Also assume that the mth power of a degree n polynomial can be computed in size $O(nm \log n)$ and depth $O(\log n)$. The algorithm for this problem should have size $O(PM(n))$ and depth $O(\log n \log\log n)$.

1.8 For this chapter's goal of taking reciprocals, the complexity of the powering circuit described in corollary 1 was sufficient; however, if the end goal is powering, substantially more efficient circuits can be designed. This problem will lead you through the steps to design such a powering circuit.

a) If you are given values x^{2^k}, for $k = 1, \cdots, \lfloor \log m \rfloor$, design a circuit for computing x^m that has $O(nm \log nm \log\log nm \log\log m)$ size and $O(\log nm \log\log m)$ depth.

b) If $m = 2^p$ for some integer p, then we can compute x^m by letting $f = \lfloor p/2 \rfloor$ and $c = \lceil p/2 \rceil$, and then noticing that

$$x^m = x^{2^p} = \left(x^{2^f}\right)^{2^c} .$$

If we use the circuit from corollary 1 for each powering, what is the complexity of this powering algorithm?

c) Part (b) can be generalized by letting $f = \lfloor p/c \rfloor$ and $c = \lceil p/c \rceil$ for any integer c. Then by combining appropriate numbers of powerings to the 2^f power and to the 2^c power, we can very efficiently compute x^{2^p}. Give the details of this generalized algorithm, and derive its complexity.

d) For the final result, use part (c) to compute the powers required as input to part (a), and combine these results to give an efficient powering circuit. For any constant $\epsilon > 0$, you should be able to derive a circuit with size $O(nm^{1+\epsilon} \log n \log\log n)$ and depth $O(\log nm \log\log m)$.

Bibliography

[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(2):447–460, April 1988.

[3] P. W. Beame, S. A. Cook, and H. J. Hoover. Log depth circuits for division and related problems. *SIAM J. Comput.*, 15(4):994–1003, November 1986.

[4] D. Bini and V. Pan. Polynomial division and its computational complexity. *J. of Complexity*, 2:179–203, 1986.

[5] D. Bini and V. Pan. Improved parallel polynomial division and its extensions. *Proc. 33rd Annual IEEE Symposium on Foundations of Computer Science*, pages 131–136, 1992.

[6] H. Chen. Private communication, 1993.

[7] S. A. Cook. *On The Minimum Computation Time of Functions*. PhD thesis, Harvard University, Cambridge, MA, 1966.

[8] W. Eberly. Very fast parallel matrix and polynomial arithmetic. *Proc. 25th Annual IEEE Symposium on Foundations of Computer Science*, pages 21–30, 1984.

[9] J. Hastad and T. Leighton. Division in $O(\log n)$ depth using $O(n^{1+\epsilon})$ processors, 1986. Unpublished note.

[10] K. Melhorn and F. P. Preparata. Area-time optimal division for $T = \Omega((\log n)^{1+\epsilon})$. *Symposium on Theoretical Aspects of Computer Science*, pages 341–352. Lecture Notes in Computer Science 210, Springer-Verlag, 1986.

[11] N. Pippenger. The complexity of computations by networks. *IBM J. Res. Dev.*, 31(2):235–243, March 1987.

[12] J. H. Reif. Logarithmic depth circuits for algebraic functions. *Proc. 24th Annual IEEE Symposium on Foundations of Computer Science*, pages 138–145, 1983.

[13] J. H. Reif. Logarithmic depth circuits for algebraic functions. *SIAM J. Comput.*, 15(1):231–241, February 1986.

[14] J. H. Reif and S. R. Tate. Optimal size integer division circuits. *21st STOC*, pages 264–273, 1989.

[15] A. Schönhage and V. Strassen. Schnelle multiplikation grosser zahlen. *Computing*, 7:281–292, 1971.

[16] N. Shankar and V. Ramachandran. Efficient parallel circuits and algorithms for division. *Inform. Process. Lett*, 29:307–313, 1988.