ACADEMIC PRESS

Journal of Computer and System Sciences 65 (2002) 695–716

JOURNAL OF COMPUTER AND SYSTEM SCIENCES

www.academicpress.com

Uniform constant-depth threshold circuits for division and iterated multiplication

William Hesse,[a,1] Eric Allender,[b,*,2] and David A. Mix Barrington[a,3]

[a] *Department of Computer Science, University of Massachusetts, Amherst, MA 01003-4610, USA*

[b] *Department of Computer Science, Rutgers University, Piscataway, NJ 08854-8019, USA*

Received 7 August 2001; received in revised form 10 June 2002; accepted 19 June 2002

Abstract

It has been known since the mid-1980s (SIAM J. Comput. 15 (1986) 994; SIAM J. Comput. 21 (1992) 896) that integer division can be performed by poly-time uniform constant-depth circuits of MAJORITY gates; equivalently, the division problem lies in P-uniform TC^0. Recently, this was improved to L-uniform TC^0 (RAIRO Theoret. Inform. Appl. 35 (2001) 259), but it remained unknown whether division can be performed by DLOGTIME-uniform TC^0 circuits. The DLOGTIME uniformity condition is regarded by many as being the most natural notion of uniformity to apply to small circuit complexity classes such as TC^0; DLOGTIME-uniform TC^0 is also known as FOM, because it corresponds to first-order logic with MAJORITY quantifiers, in the setting of finite model theory. Integer division has been the outstanding example of a natural problem known to be in a P-uniform circuit complexity class, but not known to be in its DLOGTIME-uniform version.

We show that indeed division is in DLOGTIME-uniform TC^0. First we show that division lies in the complexity class FOM + POW obtained by augmenting FOM with a predicate for powering modulo small primes. Then we show that the predicate POW itself lies in FOM. (In fact, it lies in FO, or DLOGTIME-uniform AC^0.)

The essential idea in the fast parallel computation of division and related problems is that of Chinese remainder representation (CRR)—storing a number in the form of its residues modulo many small primes. The fact that CRR operations can be carried out in log space has interesting implications for small space classes. We define two versions of $s(n)$ space for $s(n) = o(\log n)$: dspace($s(n)$) as the traditional version where the worktape begins blank, and DSPACE($s(n)$) where the space bound is established by endmarkers before the computation starts. We present a new translational lemma characterizing the unary languages in the DSPACE classes. It is known (Theoret. Comput. Sci. 3 (1976) 213) that $\{0^n : n \text{ is prime}\} \notin \text{dspace}(\log\log n)$. We show that if this can be improved to $\{0^n : n \text{ is prime}\} \notin \text{DSPACE}(\log\log n)$, it follows that $L \neq NP$.

*Corresponding author.

E-mail addresses: whesse@cs.umass.edu (W. Hesse), allender@cs.rutgers.edu (E. Allender), barring@cs.umass.edu (D.A. Mix Barrington).

[1] Supported by NSF Grant CCR-9877078.

[2] Supported in part by NSF Grants CCR-9734918 and CCR-0104823.

[3] Supported in part by NSF Grant CCR-9988260.

PII: S0022-0000(02)00025-9

Keywords: Division; Iterated multiplication; Uniform threshold circuits; Circuit complexity; Chinese remainder representation; Computation in abelian groups

1. Introduction

In 1984, Beame et al. [15] presented new parallel algorithms for division, powering, and iterated multiplication of integers. They showed that these problems can be solved by families of circuits with fan-in two and $O(\log n)$ depth, placing them in the circuit class NC^1. As Reif [46,47] observed soon after, their algorithms can also be implemented by families of *threshold circuits* with constant depth and polynomial size, placing these problems in the circuit class TC^0. Equivalently [17], these problems are *reducible* in a strong sense to ordinary integer multiplication—they can be solved by constant-depth poly-size circuits of multiplication gates.

The outstanding issue remaining after Beame et al. [15] was the degree of *uniformity* of the circuit families for division and iterated multiplication. The circuits of Beame et al. are "P-uniform", meaning that the nth circuit can be constructed by a poly-time Turing machine from the number n given in unary. Thus they are apparently more difficult to construct than to evaluate (for example, they were not known to be constructible by a logspace Turing machine and hence did not serve to divide numbers in logspace). In a recent breakthrough, Chiu et al. [19] developed logspace uniform circuits and thus showed these problems to be in L-uniform TC^0. They thus also solved an even older open problem by showing these problems to be solvable in L itself. (L is the class of functions that can be computed by deterministic Turing machines using logarithmic space.)

If (as is widely believed) L is different from TC^0, the work of Chiu et al. [19] still yields circuits that are more difficult to construct than to evaluate. There is a natural definition of "fully uniform TC^0", which is robust across several different models of computation [13]. Two formulations of this class are DLOGTIME-uniform circuits and problems definable by first-order formulas with MAJORITY quantifiers. Are division, powering, and iterated multiplication computable within this class?

We resolve this question affirmatively. All three of these problems can be solved by fully uniform families of threshold circuits of constant depth and polynomial size. Equivalently, all three are reducible to integer multiplication by fully uniform circuits of constant depth and polynomial size.

There are two parts to our proof. First (in work first reported in [6]), we show that the non-uniformity necessary for the construction of [19] is quite limited: In Immerman's descriptive complexity setting [36], we need only first-order formulas with MAJORITY quantifiers and a single extra numerical predicate. This predicate expresses powering of integers modulo a prime of $O(\log n)$ bits.

Next (in work first reported in [35]), we show that powering modulo any small prime is in DLOGTIME-uniform AC^0. (Equivalently, it can be expressed in first-order logic on ordered structures, with addition and multiplication.)

We also consider the implications of the new division algorithm for the study of small-space complexity classes. Most prior work on Turing machines with $O(\log \log n)$ space, for example, has assumed that the work tape starts out blank, with no marker to indicate the end of the available space. We call this class

dspace($\log\log n$), in contrast to the class DSPACE($\log\log n$) where this initial marker is given.

The space-efficient CRR algorithms allow us to prove more efficient translational arguments, showing that the *unary* languages in DSPACE($\log\log n$) are simply the unary encodings of the languages in log space. This highlights the difference between dspace and DSPACE classes. For example, a classic result of Hartmanis and Berman [30] says that the set of unary strings of prime length is not in dspace($o(\log n)$). The new translational lemma shows that proving an analogous result for DSPACE($\log\log n$) would separate the classes L and NP.

In Section 2 we present the necessary definitions and background. In Section 3 we review the history and context of the numeric problems we study. In Section 4 we outline an algorithm based on that of Chiu et al. [19], showing that the necessary CRR operations for division can be carried out by circuits that are uniform, given a predicate for powering modulo small numbers. In Sections 5 and 6 we show that this predicate is in DLOGTIME-uniform AC^0. Finally, our results on small-space-bounded classes are presented in Section 7.

2. Definitions

We are concerned with the complexity of three basic problems in integer arithmetic (with input and output in binary representation):

- DIVISION: Given a number X of n bits and a number Y of at most n bits, find $\lfloor X/Y \rfloor$,
- POWERING: Given a number X of n bits and a number k of $O(\log n)$ bits, find X^k, and
- ITERATED MULTIPLICATION: Given n numbers $X_1, \ldots, X_n$, each of at most n bits, find the product $X_1 X_2 \ldots X_n$.

A number of our formal definitions require that we consider these problems to be formally given as *predicates* rather than as functions. Thus, for example, we will express division as a predicate DIVISION(X, Y, i) which is true if and only if bit i of $\lfloor X/Y \rfloor$ is 1. Similarly, the iterated multiplication problem will be written as the predicate IMULT$(A_1, \ldots, A_n, i)$ which is true if bit i of $\prod_{j=1}^{n} A_j$ is 1; i ranges from 0 to $n^2 - 1$, and so has $2\log n$ bits.

We denote numbers with n or $n^{O(1)}$ bits by uppercase letters, and numbers with $O(\log n)$ bits by lowercase letters. We also refer to numbers with $O(\log n)$ bits as short, and those with $n^{O(1)}$ bits as long. We will always note the size of numbers with $\log^{O(1)} n$ bits explicitly.

Note that all the circuit complexity classes and descriptive complexity classes we consider are closed under a polynomial change in the input size. Therefore, though the size of the input to division is $2n + \log n$ and that of the input to iterated multiplication is $n^2 + 2\log n$, we will consider the input size to be n for all problems in this paper.

2.1. Circuit classes

We begin by formally defining the three circuit complexity classes that will concern us here. These are given by *combinatorial* restrictions on the circuits of the family. We will then define the uniformity restrictions we will use. Finally, we will give the equivalent formulations of uniform circuit complexity classes in terms of descriptive complexity classes.

NC^1 is the class of languages A for which there exist circuit families $\{C_n : n \in \mathbb{N}\}$ where each circuit C_n

- computes the characteristic function of A on inputs of length n,
- consists of AND and OR gates of fan-in two, and NOT gates,
- has depth $O(\log n)$ (and consequently has size $n^{O(1)}$).

TC^0 is the class of languages A for which there exist circuit families $\{C_n : n \in \mathbb{N}\}$ where each circuit C_n

- computes the characteristic function of A on inputs of length n,
- consists of MAJORITY gates (with no bound on the fan-in), and NOT gates,
- has depth $O(1)$,
- has size $n^{O(1)}$.

AC^0 is the class of languages A for which there exist circuit families $\{C_n : n \in \mathbb{N}\}$ where each circuit C_n

- computes the characteristic function of A on inputs of length n,
- consists of AND and OR gates (with no bound on the fan-in), and NOT gates,
- has depth $O(1)$,
- has size $n^{O(1)}$.

It will cause no confusion to use the terms NC^1 and TC^0 also to refer to classes of *functions* computed by these classes of circuits, instead of merely focusing on *languages*.

2.2. Uniformity

The circuit classes NC^1, TC^0, and AC^0 each come in different flavors corresponding to different *uniformity conditions*. As defined above, these classes are *non-uniform*. That is, there is *no restriction* on how difficult it is to compute the function $n \mapsto C_n$ (i.e., on how hard it is to *build* the circuits). In order to make circuit classes comparable to standard sequential complexity classes such as P and L, we need to place such restrictions.

In particular, our three circuit classes are only known to be contained within P if they are *P-uniform*, meaning that the function $n \mapsto C_n$ must be computable in polynomial time. They are only known to become subclasses of L if we make them L-*uniform* by making this function computable in L. But even L-uniformity is unsatisfactory in that the L-uniform versions of these classes apparently contain languages that are solvable because of the capabilities of the machine constructing the circuit rather than those of the circuit itself.

Barrington et al. [13] proposed a robust definition of *fully uniform* versions of each of these classes. Following Ruzzo [49], the condition requires that a DLOGTIME Turing machine (with random access to its input) be able to answer particular questions about each circuit, such as "is gate i of the n-input circuit a child of gate j". (More complex questions are needed in the case of NC^1 for technical reasons [13,49].)

A consensus has developed among researchers in circuit complexity that this DLOGTIME uniformity is the "right" uniformity condition. This argument is made in some detail in [12]—in particular:

- DLOGTIME uniformity is equivalent to P or L uniformity *except* in the case of small circuit complexity classes; i.e. in those cases where the circuits are not capable of simulating the machine constructing them,
- All the standard constructions relating circuits to sequential models can be carried out with DLOGTIME uniform circuits, and most importantly,
- The new complexity classes defined in terms of DLOGTIME uniform circuits have many equivalent characterizations in terms of other models such as alternating Turing machines, first-order logic, and functional algebra [7,13,20].

Thus **for the rest of this paper, any reference to "uniform" circuits means "DLOGTIME-uniform" circuits, unless some other uniformity condition is explicitly mentioned.** Similarly, any mention of AC^0, TC^0, or NC^1 will refer to the DLOGTIME-uniform versions of these classes, unless another uniformity condition is explicitly mentioned.

We will make use of some standard parallel algorithms in arithmetic, all of which can be carried out by DLOGTIME-uniform circuits. For example, in AC^0 one can add two n-bit numbers and determine the number of ones in a binary string of length $\log^{O(1)} n$. (In fact one can also determine the number of ones in a string of length n if this number is $\log^{O(1)} n$.) In TC^0 one can multiply two n-bit numbers, and add together n n-bit numbers. For background on these algorithms, the reader can consult [13,17,24,34,36,44].

Only one strict containment relation is known among the complexity classes that are mentioned above. It is known that $AC^0 \subseteq TC^0 \not\subseteq NC^1 \subseteq L \subseteq P$.

2.3. Descriptive complexity classes

Our goal in this paper is to show that DIVISION and ITERATED MULTIPLICATION are in the class DLOGTIME-uniform TC^0. Applying the definition directly, this would require us to describe a family of constant-depth polynomial-size threshold circuits and then argue that they meet the uniformity condition. Instead, we will make use of one of the alternate characterizations of DLOGTIME-uniform circuit classes given above, that of *descriptive complexity*.

A problem is in the complexity class FO (first-order) if the predicate corresponding to the decision problem can be expressed by a first-order formula interpreted over a finite universe: the set of natural numbers $0, \ldots, n-1$. The inputs to a problem are encoded as relations (i.e., X, Y, etc.) over the universe. Formulas are formed in the usual way using universal and existential quantifiers and the relations X, Y, etc., as well as the numeric relations[4] $<$ and BIT. The problems in FO coincide exactly with the problems in the complexity class DLOGTIME-uniform AC^0 [13].

The complexity class FOM (first-order with MAJORITY) is defined analogously, except that MAJORITY quantifiers are allowed, in addition to the usual universal and existential quantifiers. The majority quantifier (Mx) can appear anywhere that an $(\exists x)$ or a $(\forall x)$ can appear. The formula $(Mx)\varphi(x)$ is true iff $\varphi(j)$ is true for more than half the values $0 \leqslant j < n$. The classes FOM and DLOGTIME-uniform TC^0 are equal [13].

Some examples may help clarify the situation. Consider the DIVISION problem. The n bits of the input X to DIVISION are represented by the values of a unary

[4] Following [36], we consider FO to include ordering and BIT. The BIT predicate allows us to look at the bits of numbers. $\mathrm{BIT}(i, x)$ is true if bit i of the number x written in binary is 1. This is equivalent in expressive power to having addition and multiplication on numbers between 0 and $n-1$.

predicate $X(\cdot)$ on the elements of the universe: $X(0), X(1), \ldots, X(n-1)$. An n^2 bit input can be represented by a binary predicate, so the inputs $A_0, \ldots, A_{n-1}$ to IMULT are represented as a binary predicate $A(\cdot,\cdot)$. Short inputs to a problem, such as i, the index of the result bit queried, may be represented by a constant in the range $0, \ldots, n-1$.

The predicate BITSUM(x, y), which is defined to be true if the binary representation of x contains y ones, is definable in FO [13,36].

Since an FO or FOM formula over the universe $0, \ldots, n^k - 1$ can be simulated by an equivalent formula over the universe $0, \ldots, n-1$, DIVISION and IMULT with inputs X, Y, and A_i having n^k bits, encoded by k-ary relations over $0, \ldots, n-1$, are in the same descriptive complexity class as DIVISION and IMULT with n-bit inputs.

This methodology will afford us two key advantages. First, the notion of defining one problem in terms of another, along with simple operations, is a very natural one. The circuit components naturally constructed for arithmetic operations are very regular in form, and the logical descriptions of them can capture this without worrying about uniformity machines. Secondly, adding more atomic predicates to the logical formalism allows for very fine distinctions in the amount of non-uniformity added to the circuits. The class FOM + POW defined in Section 4 was a key step toward our eventual result, and this class would have been considerably harder to define without the first-order logic formalism.

In order to present our results, we frequently make use of the notion of "FO-Turing reducibility" between problems. This is formally defined using generalized quantifiers in [36]. For our applications, this notion is equivalent to the notion of DLOGTIME-uniform AC^0-Turing reducibility, as described in [55] (using "oracle gates"). In all cases, when we say that A is FO-Turing-reducible to B, it should be clear that if B is in DLOGTIME-uniform AC^0, then so is A.

For further background on descriptive complexity, please consult Immerman's monograph [36].

3. Circuits for division: an overview

Beame et al. [15] showed that DIVISION, POWERING, and ITERATED MULTIPLICATION are all easily reducible to one another. We find it convenient to work mostly with ITERATED MULTIPLICATION in this paper, but for completeness we review here why uniform circuits for ITERATED MULTIPLICATION provide uniform circuits for DIVISION.

Since there is a convergent power series for the reciprocal of a real number, we can use iterated multiplication to approximate that power series. Since $1/(1-\alpha) = \sum_{i=0}^{\infty} \alpha^i$, for any real number α with $|\alpha| < 1$, then for $1/2 < \alpha \leqslant 1$,

$$1/\alpha = \sum_{i=0}^{n} (1-\alpha)^i + O(2^{-n}).$$

To divide X by Y, let $j = \lceil \log Y \rceil$ be roughly the number of bits in Y. Then use $2^{-j}Y$, which is between $\frac{1}{2}$ and 1, as α in the preceding formula. Multiplying by 2^{nj} to create an integer computation, we find

$$2^{nj}X/Y = X \sum_{i=0}^{n} (2^j - Y)^i (2^j)^{n-i} + O(X 2^{nj-n}).$$

Since the error is $O(2^{nj})$, we find $\lfloor X/Y \rfloor$ to within an additive error of $O(1)$ by dropping the lowest nj bits of the result. The exact value of $\lfloor X/Y \rfloor$ can be found by multiplying our approximation by Y and comparing to X.

Since addition of polynomially many numbers can be performed in TC^0, this entire algorithm can be viewed as an efficient reduction from DIVISION to POWERING, which is a special case of ITERATED MULTIPLICATION.

For a more detailed exposition, consult [15], where it is also observed that this technique allows the approximate computation of any function given by a convergent power series.

It was shown in [15] that ITERATED MULTIPLICATION is in P-uniform NC^1. It was observed later by Reif [46,47] that the same algorithm can be implemented in P-uniform TC^0. It was also noticed that the construction is logspace uniform, given access to the product of the first n^3 primes [15,37], and hence the full construction is TC^1 uniform. There has also been work reducing the size and depth of division circuits. Division circuits of depth $O(\log n)$ and size $n^{1+\varepsilon}$ were presented in [33,48,50]. Polynomial-size MAJORITY circuits of depth three are known for DIVISION and POWERING; depth four suffices for ITERATED MULTIPLICATION [51]. On the related model of circuits with AND, OR, and MAJORITY gates, it is known that two layers of MAJORITY gates suffice to compute DIVISION and POWERING; three layers suffice for ITERATED MULTIPLICATION [41].

It remained unknown whether DIVISION could be computed in logarithmic space, although an algorithm using nearly-logarithmic space was presented already by Reif in the 1980s [45]. This situation was remedied when Chiu et al. [19] presented an improved algorithm that can be implemented in L-uniform TC^0.

Chiu's Master's thesis [18] also shows that division lies in fully uniform NC^1.

In the next section, we present a simplified division algorithm that was inspired by Chiu et al. [19]. Our presentation is in terms of descriptive complexity, but can equally well be thought of explicitly in terms of circuits. Each step is a description of a parallel computation of some predicate in terms of previously computed predicates and basic operations that can be expressed by quantifiers or (equivalently) clearly be performed by polynomial size parallel circuits with a simple structure. For example we can say "Compute the product p^2 for each prime p less than n^2". This is simple because deciding whether a short number is prime is in FO.

Since our prime focus is on potential non-uniformity in the circuit, we must take note when we need inputs that are not obviously computable. In particular, we will need to use the values $a^i \bmod p$ for short values a, i, and p. In descriptive complexity terms, we will refer to the fixed numeric predicate POW, where

$$\mathrm{POW}(a,i,b,p) \Leftrightarrow a^i \equiv b \pmod{p}$$

and all of the inputs have $O(\log n)$ bits. This is represented in logic as a fixed relation of arity $4k$ and size n^{4k}, assuming the inputs have $k \log n$ bits.

We define the descriptive complexity classes $\mathrm{FO}+\mathrm{POW}$ and $\mathrm{FOM}+\mathrm{POW}$ to be FO and FOM, respectively, augmented with this new atomic predicate POW. Just as FO and FOM are DLOGTIME-uniform versions of AC^0 and TC^0, these new classes are "slightly less uniform" versions of the same classes, in that answering questions about the circuits might require access to powers of short numbers. (We will sometimes call such circuits "POW-uniform".) As we will eventually show, the predicate POW is *itself* computable in FO (and thus also in FOM), so in fact $\mathrm{FO}+\mathrm{POW}$ and $\mathrm{FOM}+\mathrm{POW}$ collapse to FO and FOM, respectively.

4. Division is in FOM + POW

The central idea of all the TC^0 algorithms for ITERATED MULTIPLICATION and related problems is that of *Chinese remainder representation* (*CRR*). An n-bit number is uniquely determined by its residues modulo polynomially many primes, each of $O(\log n)$ bits. The Prime Number Theorem guarantees that there will be more than enough primes of that length.

To fix notation, we now recapitulate the development of CRR. If we are given a sequence of distinct primes $m_1, \ldots, m_k$, each a short number, let M be their product. Any number $X < M$ can be represented uniquely as $(x_1, \ldots, x_k)$ with $X \equiv x_i \pmod{m_i}$ for all i. For each number i, let C_i be the product of all the m_j's except m_i, and let h_i be the inverse of C_i modulo m_i. It is easy to verify that X is congruent modulo M to $\sum_{i=1}^{k} x_i h_i C_i$. In fact X is *equal*, as an integer, to $(\sum_{i=1}^{k} x_i h_i C_i) - rM$ for some particular number r, called the *rank* of X with respect to M (denoted $\mathrm{rank}_M(X)$). Note that r is a short number. It is equal to the integer part of the sum of the k rational numbers $x_i h_i C_i / M$ or $x_i h_i / m_i$, each of which is between 0 and m_i.

The algorithm for ITERATED MULTIPLICATION is easy to describe:

Step 1: Convert the input from binary to CRR.
Step 2: Compute the iterated product in CRR.
Step 3: Convert the answer from CRR to binary.

Step 1 is easy to accomplish in FOM + POW. A proof is provided in Lemma 4.1.

Step 2 is solved by adding discrete logarithms. The following few paragraphs explain this in more detail.

Since the multiplicative group $\mathbb{Z}_p^*$ of a prime number is cyclic, of order $p-1$, the predicate POW allows us to identify the smallest generator of this group. This is the least g such that $g^i \not\equiv 1 \pmod{p}$ for $0 < i < p-1$. This implies that the values g^i are all distinct for $0 \leqslant i < p-1$, and that $g^i \equiv a \pmod{p}$ has a unique solution for each $0 < a < p$; this number i is known as the *discrete logarithm* of a. There is clearly an FO + POW formula $\mathrm{GEN}(g,p)$ stating that g is the smallest generator of $\mathbb{Z}_p^*$, and an equivalent POW-uniform AC^0 circuit computing g from p. Remember that p and g are short numbers, with $O(\log n)$ bits.

Similarly, there is an FO + POW formula $\mathrm{GEN}(g,p)$ AND $\mathrm{POW}(g,i,a,p)$ stating that i is the discrete logarithm of a.

Now note that *if the input and output are in CRR*, the iterated multiplication problem simply reduces to the iterated addition problem (by adding the discrete logs). That is, in order to compute $\prod A_i \bmod p$, where A_i is equivalent to $g^{\ell_i} \bmod p$, we simply compute $b = \sum \ell_i$ and output $g^b \bmod p$—which is easy to do in FOM + POW.

Steps 1 and 2 are essentially identical to the initial part of the construction that was used in [15].

Step 3 (converting from CRR to binary) requires additional work. In order to convert from binary to CRR, Beame et al. needed an additional predicate: the binary representation of the product of the first n^3 primes. While the power predicate is easily seen to be computable in logspace, this prime-product predicate was not known to be so easy to compute. The central contribution of Chiu et al. [19] was to develop better methods for working with CRR, so that the prime-product predicate is no longer needed. In this section, we present a procedure for conversion from CRR to binary that can be computed in FOM + POW. Thus the power predicate, the essential ingredient in converting a binary number *into* CRR, is powerful enough (along with FOM operations) to get a number *out of* CRR into binary.

The computation of the rank function is central to the argument of Chiu et al. [19] that DIVISION is in L-uniform TC^0. It is computable in logspace [23,40], and in fact the algorithms can be adapted to put it in FOM + POW. (For more detail on this see [5].) Here we present a self-contained argument, without computing rank directly, that conversion from CRR to binary is in FOM + POW.

First we note again that we can carry out the other conversion, from binary to CRR.

Lemma 4.1. *If $X, m_1, \ldots, m_k$ are each given in binary and $X < M$, we can compute $(x_1, \ldots, x_k)$ (the CRR_M form of X) in* FOM + POW.

Proof. For each modulus m_i and each $j < n$ we must calculate $2^j \bmod m_i$ (given by the power predicate), add the results (using iterated addition in FOM), and take the result modulo m_i (in FO). □

It will be useful to observe that dividing by a short prime is easy.

Lemma 4.2. *Let p be a short prime. Then the binary representation of $1/p$ can be computed to $n^{O(1)}$ bits of accuracy in* FO + POW.

Proof. Let p be odd and write 2^s as $ap + b$ with $b = 2^s \bmod p$. The sth bit of the binary expansion of the rational number $1/p$ is equal to the low-order bit of a. Since $ap + b$ is congruent to zero modulo 2, and since p is odd, it follows that the low-order bit of a is also the low-order bit of b. Since b is $2^s \bmod p$, it can clearly be computed in FO + POW. □

Lemma 4.3 (Davida and Litow [23], Dietz et al. [25]). *Let X and Y be numbers less than M given in CRR_M form. In* FOM + POW *we can determine whether $X < Y$.*

Proof. Clearly, $X < Y$ if and only if $X/M < Y/M$. Thus it is sufficient to show that we can compute X/M to polynomially many bits of accuracy.

Recall that $X = (\sum_{i=1}^{k} x_i h_i C_i) - \mathrm{rank}_M(x)M$. Thus X/M is equal to $(\sum_{i=1}^{k} x_i h_i (1/m_i)) - \mathrm{rank}_M(x)$. The numbers x_i are given to us as the CRR_M of X. The number $C_i \bmod m_i$ can be computed in FOM + POW (by adding the discrete logs of the m_j for $j \neq i$), and h_i is simply the inverse of that number $\bmod m_i$. By Lemma 4.2, each summand can be computed in FOM + POW to $n^{O(1)}$ bits of accuracy. Since iterated addition is in FOM, we can thus compute polynomially many bits of the binary representation of $(\sum_{i=1}^{k} x_i h_i (1/m_i))$, which is equal to $X/M + \mathrm{rank}_M(X)$. Since the rank is an integer, X/M is simply the fractional part of this value and we now have each bit of it available. □

One useful consequence of being able to compare integers in CRR is that it enables us easily to convert from one CRR basis to another. That is, if we are given X in CRR_M for one list of moduli $m_1, \ldots, m_k$, $M = \prod_{i=1}^{k} m_i$ and we want to convert to CRR_P for some list of distinct short primes $p_1, \ldots, p_l$, $P = \prod_{i=1}^{l} p_i$, all that is necessary is to compute $X \bmod p$ for an arbitrary short prime p.

Lemma 4.4. *Given X in CRR_M and a short prime p, we can compute $X \bmod p$ in* FOM + POW.

Proof. If p is one of the moduli in M, the answer is given explicitly in the input. Thus we assume that p does not divide M. In this case, consider the CRR

base $M' = Mp$. We would like to compute X in $\mathrm{CRR}_{M'}$, since this would give us $X \bmod p$.

Trying each of the $p = n^{O(1)}$ possible values i for $X \bmod p$, we obtain the $\mathrm{CRR}_{M'}$ of $n^{O(1)}$ different numbers $X_0, X_1, \ldots, X_{p-1}$, one of which is X. It is easy to see that X is the only one of these numbers that is less than M.

Observe that in FOM + POW we can compute the $\mathrm{CRR}_{M'}$ of M (by adding the discrete logs of the $m_j \bmod p$). Thus we can compute $X \bmod p$ by finding the unique X_i that is less than M, carrying out all comparisons in $\mathrm{CRR}_{M'}$, by Lemma 4.3. □

Our next step in the division algorithm is to show how to divide by products of distinct short primes.

Lemma 4.5. *Let $b_1, \ldots, b_\ell$ be distinct short primes, B be the product of the b_i's, and let X be given in* CRR_M *form. Then we can compute* $\lfloor X/B \rfloor$, *also in* CRR_M *form, in* FOM + POW.

Proof. Assume without loss of generality that B divides M. (Otherwise, extend the basis, using Lemma 4.4.) Then let $M = BP$.

In FOM + POW we can compute the following quantities:

- $X \bmod B$ in CRR_B (by dropping the primes in P from our basis),
- $X \bmod B$ in CRR_M (by extending the basis),
- $X - (X \bmod B) = B\lfloor X/B \rfloor$ in CRR_M.

Since B and P are relatively prime, there is a B^{-1} such that $B\,B^{-1} \equiv 1 \pmod P$. We can find its representation in CRR_P, by merely finding the inverse of each component relative to the (short) modulus for that component, using discrete logs.

Then

$$B^{-1}B\lfloor X/B \rfloor \equiv \lfloor X/B \rfloor \pmod P,$$

and since $X < M$, $\lfloor X/B \rfloor < P$ so we can calculate $\lfloor X/B \rfloor$ in CRR_M by calculating $B^{-1}B\lfloor X/B \rfloor$ in CRR_P and extending the basis. □

Theorem 4.1. *Let X be given in* CRR_M *form* $(0 \leqslant X < M)$. *Then we can compute the binary representation of X in* FOM + POW.

We remark that a simple extension of this result and Lemma 4.1 shows that it is possible in FOM + POW to convert numbers from any base to another, by first converting to CRR.

Proof. It is sufficient to show that we can compute the CRR_M of $\lfloor X/2^s \rfloor$ for any s. This is because, to get the sth bit of a number X that is given to us in CRR, we compute $u = \lfloor X/2^s \rfloor$ and $v = \lfloor X/2^{s+1} \rfloor$, and note that the desired bit is $u - 2v$. We get this bit as a CRR number, but it is easy to recognize the CRR forms of the numbers 0 and 1.

First, we create numbers $A_1, \ldots, A_s$, each a product of polynomially many distinct short odd primes that do not divide M, with each $A_i > M$, and A_i relatively prime to A_j for $i \neq j$. Here is how we create the A_i. Recall that $M = \prod_{j=1}^{k} m_i$; assume that m_k is the largest of the prime factors of M. Note that the product of any consecutive k larger primes is larger than M. Thus each A_i can be taken to be $\prod_{j=1}^{k} p_{ik+j}$ (where $p_{k+1}, p_{k+2}, \ldots$ is the list of consecutive primes larger than m_k). The list of primes less

than $n^{O(1)}$ can be computed by an FOM circuit; the prime number theorem guarantees that there are enough primes on this list for our needs.

Our approach to computing $\lfloor X/2^s \rfloor$ is to note that dividing X by 2^s is quite similar to multiplying X by $(\prod_{i=1}^{s} (1+A_i)/2)/(\prod_{i=1}^{s} A_i)$. Let $P = M \prod_{i=1}^{s} A_i$, and compute X in CRR_P. By Lemma 4.5 (or directly) we can compute the integer $(1+A_i)/2$ in CRR_P. It is easy to show that $(\prod_{i=1}^{s} (A_i+1))/\prod_{i=1}^{s} A_i < (1+1/M)^s$, and in turn this is less than $1+(s+1)/M$, since $s < \log M \ll M$.

Note that in FOM + POW (using Lemma 4.5) we can compute the CRR_P representation of $Q = \lfloor X \prod_{i=1}^{s} ((1+A_i)/2)/\prod_{i=1}^{s} A_i \rfloor$. But

$$X \prod_{i=1}^{s} ((1+A_i)/2)/\prod_{i=1}^{s} A_i$$
$$= (X/2^s)\left(\prod_{i=1}^{s} (A_i+1))/\prod_{i=1}^{s} A_i < (X/2^s)(1+((s+1)/M)\right).$$

Thus $Q \in \{\lfloor X/2^s \rfloor, \lfloor X/2^s \rfloor + 1\}$. We determine which of $\{Q, Q-1\}$ is the correct answer by checking if $Q2^s > X$ (using the CRR_P representation). □

Corollary 4.1. DIVISION, ITERATED MULTIPLICATION, *and* POWERING *are all in* FOM + POW.

5. Two special cases in FO

This section will prove two partial steps toward our main theorem, which will be used in the proof. We will show that POW, IMULT, and DIVISION are all in FO when the size of the inputs is reduced exponentially.

Lemma 5.1. POW(a, r, b, p), *where the inputs have* $O(\log\log n)$ *bits, is in* FO.

Proof. If a, r, b, and p all have $k \log\log n$ bits, then we can compute $a^r \bmod p$ in FO using repeated squaring. Consider the sequence of exponents $r_0, r_1, \ldots, r_{k \log\log n}$, where $r_i = \lfloor r/2^i \rfloor$. Then $r_0 = r$ and $r_{k \log\log n} = 0$. Also, $r_i = 2r_{i+1}$ or $r_i = 2r_{i+1}+1$, depending on the corresponding bit of r.

We will simultaneously guess the numbers $a_i = a^{r_i} \bmod p$, and verify that they obey the conditions $a_{k \log\log n} = 1$ and $a_i = a_{i+1}^2 \bmod p$ or $a_i = a_{i+1}^2 a \bmod p$, depending on the bits of r. We do this by checking all possible combinations of the a_i in parallel. Since the $k \log\log n$ numbers a_i each have $k \log\log n$ bits, there are $2^{k^2(\log\log n)^2}$ possible ways of choosing these bits. This is asymptotically fewer than n possibilities, so we need to do fewer than n computations in parallel. The a_i can be encoded into the $\log n$ bits of a number between 1 and n in a simple way, so the verification that one of these guesses for the sequence a_i is correct, and that $a_0 = b$ for this sequence, can be done by an FO formula. □

Theorem 5.1. IMULT *and* DIVISION, *where the inputs have* $(\log n)^{O(1)}$ *bits, are in* FO.

Proof. We have shown in Section 4 that IMULT and DIVISION with inputs of size r can be expressed by FOM + POW formulae over the universe $0, \ldots, r-1$. If we set $r = (\log n)^k$, then we see that IMULT and DIVISION with inputs of size $(\log n)^k$ can be expressed by FOM + POW formulae over the universe $0, \ldots, (\log n)^k - 1$. We show

that these FOM + POW formulae are equivalent to FO formulae over the universe $0, \ldots, n-1$.

Note that all uses of POW in these formulae are called with inputs of $O(\log(\log n)^k) = O(\log \log n)$ bits. Therefore, by Lemma 5.1, these uses of POW can be replaced by FO formulae with the quantified variables in the range $0, \ldots, n-1$. The uses of majority quantifiers in the FOM + POW formula, where the quantified variable is in the range $0, \ldots, (\log n)^k - 1$, can be replaced by expressions using universal and existential quantifiers over the range $0, \ldots, n-1$. This is because (as we noted earlier) counting the number of ones in an input of polylogarithmically many bits is in FO [24,28].

In this way, an FOM + POW formula over the universe $0, \ldots, (\log n)^k - 1$ can be transformed into an equivalent FO formula over the universe $0, \ldots, n-1$. □

This theorem will be needed for our proof that POW is in FO. It is also a worthwhile result in it own right, and gives a *tight* bound on the size of IMULT and DIVISION problems that are in FO. Since finding the parity of a bit string of length $f(n)$ is in FO (with universe of size n) if and only if $f(n) = (\log n)^{O(1)}$ [28,32], and parity of $f(n)$ bits reduces to IMULT of $f(n)$ numbers [17], we see that IMULT of $f(n)$ numbers is not in FO if $f(n) = (\log n)^{\omega(1)}$. We proved above that it is in FO if $f(n) = (\log n)^{O(1)}$, so the two bounds match.

6. The final step: POW is in FO

In this section, we prove our main results. First we prove a general result, showing that powering in any group is efficiently reducible to the problem of multiplying a small number of group elements. This enables us to prove that POW is FO-reducible to instances of DIVISION and IMULT with inputs of size $(\log n)^{O(1)}$. By Theorem 5.1, this implies that POW is in FO. Since we showed DIVISION and IMULT to be in FOM + POW (Corollary 4.1) this implies that they are also in FOM itself. At the end of the section, we consider a variety of applications of these results.

6.1. Powering in groups

To show that POW is in FO, we will prove a more general lemma about finding powers in groups (in the spirit of [14]). This is interesting in its own right, and necessary for the extension to finding powers modulo prime power moduli. We consider a group to be given in FO if group elements are labeled by elements of the universe and the product operation is given by an FO formula. Note that the identity element and inverse operation can be defined in FO from the product operation. We can also continue to use arithmetic operations on the universe, considered as the numbers $0, \ldots, n-1$.

Lemma 6.1. *Finding small powers in any group of order n is* FO*-Turing-reducible to finding the product of* $\log n$ *elements.*

Proof. Suppose we want to find a^r, where a is an element of a group of order n. We will compute a set of elements $a_1, \ldots, a_k$ and exponents u, $u_1, \ldots, u_k$ (with $k = o(\log n)$) such that

$$a^r = a^u a_1^{u_1}, \ldots, a_k^{u_k}$$

and $u_i < 2\log n$, $u < 2(\log n)^2$. The important thing to note is that we will compute each a_i by taking the product of a small number of group elements, and then each term in the final product is also obtainable using a small number of multiplications.

Our overall strategy for identifying the a_i and u_i is to choose the a_i to be approximations to dth roots of unity for small primes d. These roots are well distributed in the multiplicative subgroup generated by a; hence any power a^r can be approximated by multiplying (small) powers of the a_i. The exponents u_i are calculated by doing Chinese remaindering on the space of exponents.

Step 1: We find a CRR basis D of primes, each of which is $O(\log n)$, such that $D > n$, the order of the group. More precisely, we choose a set of $k = o(\log n)$ primes $d_1, \ldots, d_k$, such that $d_i < 2\log n$ and d_i is relatively prime to n, for all i. We choose them such that $n < D = d_1, d_2, \ldots, d_k < n^2$. We can do this with a first order formula by choosing the first $D > n$ such that D is square-free, D and n are relatively prime, and all prime factors of D are less than $2\log n$. We can decide, given D, whether a number is one of our d_i or not. To compute the number k from D, and to find our list d_i as a relation between i and d_i, requires, for each prime $p_0 < 2\log n$, counting the number of primes p dividing D which are less than p_0. We can do this using the BITSUM predicate.

Step 2: We calculate $a_i = a^{\lfloor n/d_i \rfloor}$ as follows:

We first compute a^{-1} using the inverse operation. Next we calculate $n_i = n \bmod d_i$ in FO. We find a^{-n_i} by multiplying n_i copies of a^{-1} together. This is one place where our Turing reduction to multiplication of $\log n$ group elements is used.

We can find $a^{\lfloor n/d_i \rfloor}$ by observing that

$$(a^{\lfloor n/d_i \rfloor})^{d_i} = a^{\lfloor n/d_i \rfloor d_i} = a^{n-(n \bmod d_i)} = a^{n-n_i} = a^{-n_i}.$$

Observe that there is exactly one group element x such that $x^{d_i} = a^{-n_i}$: Let d_i^{-1} be the multiplicative inverse to $d_i \bmod n$, i.e., that $d_i d_i^{-1} = mn + 1$ for some m. Then

$$x = x^{mn+1} = (x^{d_i})^{d_i^{-1}} = (a^{-n_i})^{d_i^{-1}}.$$

Thus we can find $a_i = a^{\lfloor n/d_i \rfloor}$ as the value of x in the expression

$$(\exists x) x^{d_i} = a^{-n_i}.$$

For each element x of the group, we compute x^{d_i} using multiplication of $\log n$ elements. We could not compute $a^{\lfloor n/d_i \rfloor}$ directly as $(a^{-n_i})^{d_i^{-1}}$ since d_i^{-1} is not necessarily $O(\log n)$.

Step 3: Now we find the exponents $u, u_1, \ldots, u_k$ such that $a^u a_1^{u_1}, \ldots, a_k^{u_k} = a^r$.

Since $a_i = a^{\lfloor n/d_i \rfloor}$, we have that

$$a_1^{u_1}, \ldots, a_k^{u_k} = a^{(\sum_{i=1}^k u_i \lfloor n/d_i \rfloor)}.$$

Since we want to obtain $u, u_1, \ldots, u_k$ such that

$$a^r = a^u a_1^{u_1}, \ldots, a_k^{u_k} = a^{(u + \sum_{i=1}^k u_i \lfloor n/d_i \rfloor)},$$

our goal can be expressed as

$$u \equiv r - \sum_{i=1}^{k} u_i \left\lfloor \frac{n}{d_i} \right\rfloor (\bmod n). \tag{1}$$

Thus, to make the final correction term a^u computable by multiplying only a small number of group elements, we must make u as small as possible. Thus we want to make $\sum_{i=1}^k u_i \lfloor n/d_i \rfloor \bmod n$ as close to r as possible.

We approximate r as a linear combination of $\lfloor n/d_i \rfloor$ in the following way. Compute $f = \lfloor rD/n \rfloor$. (This can be performed in FO since r has $O(\log n)$ bits.) Letting $D_i = D/d_i$, compute $u_i = fD_i^{-1} \bmod d_i$. ($D_i^{-1}$ can be found in FO since we can

guess possibilities for D_i^{-1} in FO.) Then, we have

$$\sum_{i=1}^{k} u_i D_i \equiv f \pmod{D}.$$

Let m be such that $\sum_{i=1}^{k} u_i D_i = f + mD$.

Calculating u from the u_i using Eq. (1) involves a sum of k short numbers, which, since $k < \log n$, is in FO. This, again, uses the fact that BITSUM is in FO.

We now show that $u < (\log n)^2$. We calculate the difference between r and $\sum u_i \lfloor n/d_i \rfloor$:

$$\begin{aligned}
\sum_{i=1}^{k} u_i \left\lfloor \frac{n}{d_i} \right\rfloor &= \sum_{i=1}^{k} \frac{u_i n}{d_i} - \sum_{i=1}^{k} \left(\frac{u_i n}{d_i} - u_i \left\lfloor \frac{n}{d_i} \right\rfloor \right) \\
&= \frac{n}{D} \sum_{i=1}^{k} u_i D_i - \sum_{i=1}^{k} u_i \left(\frac{n}{d_i} - \left\lfloor \frac{n}{d_i} \right\rfloor \right) \\
&= \frac{n}{D}(f + mD) - \sum_{i=1}^{k} u_i \left(\frac{n}{d_i} - \left\lfloor \frac{n}{d_i} \right\rfloor \right) \\
&= \frac{n}{D} \left\lfloor \frac{rD}{n} \right\rfloor + nm - \sum_{i=1}^{k} u_i \left(\frac{n}{d_i} - \left\lfloor \frac{n}{d_i} \right\rfloor \right) \\
&= r - \frac{n}{D} \left(\frac{rD}{n} - \left\lfloor \frac{rD}{n} \right\rfloor \right) + nm - \sum_{i=1}^{k} u_i \left(\frac{n}{d_i} - \left\lfloor \frac{n}{d_i} \right\rfloor \right),
\end{aligned}$$

so

$$u = r - \sum_{i=1}^{k} u_i \left\lfloor \frac{n}{d_i} \right\rfloor \bmod n = \frac{n}{D} \left(\frac{rD}{n} - \left\lfloor \frac{rD}{n} \right\rfloor \right) + \sum_{i=1}^{k} u_i \left(\frac{n}{d_i} - \left\lfloor \frac{n}{d_i} \right\rfloor \right).$$

The quantity $y - \lfloor y \rfloor$ is always between 0 and 1, and since $n/D < 1$, $u_i < 2 \log n$, and $k < \log n$, we see that $u < 2(\log n)^2 + 1$. Thus we can calculate a^u using two rounds of multiplying $\log n$ group elements.

Thus we have described group elements a_i and numbers u, u_i such that $a^u a_1^{u_1}, \ldots, a_k^{u_k} = a^r$ and the computation of $a^u a_1^{u_1}, \ldots, a_k^{u_k}$ is FO-Turing-reducible to finding products of $\log n$ group elements. □

Because FO is closed under polynomial change in input size, and the product of $\log(n^k) = k \log n$ group elements is FO-reducible to the product of $\log n$ group elements, we have

Corollary 6.1. *Finding powers in any group of order n^k* is FO-*Turing-reducible to finding the product of* $\log n$ *elements.*

Representing a group of order n^k means representing elements as k-tuples of universe elements, and representing the product operation in FO.

6.2. Powering modulo small numbers in FO

We now apply this to the integers modulo p, where $p = O(n^k)$ is a prime. The multiplicative group $\mathbb{Z}_p^*$ contains the $p - 1$ integers $1, \ldots, p - 1$, and multiplication in this group is clearly first-order definable from multiplication and addition on $0, \ldots, n - 1$. If a in POW(a, r, b, p) is zero, then we only need to check that b is zero. Otherwise, we find a^r in the multiplicative group $\mathbb{Z}_p^*$. The product of $\log n$ group

elements can be computed with IMULT and DIVISION, using inputs of size $\log^2 n$, so we have the main lemma of this section:

Lemma 6.2. *POW is FO-Turing-reducible to instances of* IMULT *and* DIVISION *with inputs of size* $(\log n)^{O(1)}$.

Corollary 6.2. POW *is in* FO.

Corollary 6.3. DIVISION *and* IMULT *are in* FOM.

Corollary 6.2 can be extended to exponentiation modulo any short number n, not just modulo a prime. We can see that the equation

$$a^r \equiv b \pmod{n}$$

is true if and only if it is true modulo all the prime power factors of n:

$$a^r \equiv b \pmod{p^i} \ \forall p^i | n.$$

We can show that for a relatively prime to p^i, a is in the group $\mathbb{Z}^*_{p^i}$, and the above proof can be applied. If p divides a, then if $r > \log n$, $a^r \equiv 0 \pmod{p^i}$. If $r \leqslant \log n$, then we can calculate $a^r \bmod n$ directly using small instances of IMULT and DIVISION. Since the prime power factors of a short number n can be found in FO, we have

Corollary 6.4. *The predicate* $a^r \equiv b \pmod{n}$, *with the inputs written in unary*, *is in* FO.

Finally, note that because any predicate expressible in FO over the universe $0, \ldots, n^k - 1$ is also expressible in FO over $0, \ldots, n-1$, we see that the predicate $a^r \equiv b \pmod{m}$ is in FO if the inputs have $O(\log n)$ bits. We cannot conclude from our present results that this predicate is in FO when the inputs have $(\log n)^{O(1)}$ bits. This contrasts with the results we have for IMULT and DIVISION with $\log^{O(1)} n$-bit inputs.

6.3. Additional applications

Efficient division circuits have found application in many settings in complexity theory. In this section we state a number of improved upper bounds that follow from the existence of DLOGTIME-uniform TC^0 circuits for division. We do not provide definitions for the problems under consideration; the reader should consult the cited references.

Corollary 6.5. *The following problems are in* DLOGTIME-*uniform* TC^0:

- *Division of Polynomials* (*with remainder*)
- *Iterated Multiplication of Polynomials*
- *Polynomial Interpolation* (*also known as Cauchy interpolation*)
- *Hermite Interpolation of Polynomials*
- *Computing* $n^{O(1)}$ *bits of* $\log X$, $X^{1/k}$, *and any other problem efficiently computed using power series approximations.*

Proof. The first three of these problems are shown in [26] to be FO-reducible to ITERATED MULTIPLICATION. (Eberly claimed only NC^1 reducibility, but his reductions are easily seen to be computable in DLOGTIME-uniform AC^0.) The

corresponding reduction for Hermite interpolation can be found in [27]. For background on approximating power series in TC^0, consult [42,47]. □

Working in the area of proof theory, Johannsen augmented the bounded arithmetic theory C_2^0 (which is closely related to FOM) with a function symbol for integer division, to obtain a class he called $C_2^0[div]$. The following are now immediate from [38].

Corollary 6.6 (Johannsen [38]).

- $C_2^0[div] = C_2^0$.
- DLOGTIME-*uniform* TC^0 *is equal to Constable's class K* [21].
- *The* Δ_1^b *theorems of* C_2^0 *do not have Craig-interpolants of polynomial circuit size, unless the Diffie–Hellman key exchange protocol is insecure.*

The complexity classes $\#AC^0$ and $GapAC^0$ were introduced in [1] and have been studied in [4,6,54]. The main motivation for introducing and studying these classes comes from the fact that they give rise to several characterizations of TC^0.

However, there was a problem with these characterizations—some of them were not known to hold in the uniform setting. For instance, four different language classes arising from arithmetic AC^0 circuits were shown in [1] to coincide with TC^0 in the non-uniform and P-uniform settings, but were not known to coincide in the DLOGTIME-uniform setting. Some more of these classes were shown to coincide in [10], but there still remained a question as to whether these classes were really the same as DLOGTIME-uniform TC^0. The answer to this question is now known.

Corollary 6.7. *All functions in* DLOGTIME-*uniform* $\#AC^0$ *can be computed in* DLOGTIME-*uniform* TC^0. *Thus the equalities* $C_=AC^0 = PAC^0 = PAC^0_{circ} = C_=AC^0_{circ} = TC^0$ *all hold in the* DLOGTIME-*uniform setting.*

Arithmetic NC^1 circuits have also been the object of considerable attention [16]; functions computed by such circuits give rise to the class $\#NC^1$. A typical function in this class is the problem of taking as input a sequence of $k \times k$ matrices of n-bit numbers (where $k = O(1)$), and computing their product. It had not been known if $\#NC^1$ functions can be evaluated in logspace. However, it is easy to show that a logspace-bounded machine can evaluate a $\#NC^1$ function modulo a small prime, and thus obtain the CRR representation of the result. Now it is also known that this CRR representation can be converted back to binary.

Corollary 6.8. *If* f *is in* $\#NC^1$, *then* f *can be computed in deterministic logspace.*

There are two well-studied ways of using a nondeterministic logspace machine to specify a function. One of these gives rise to the class FNL (which can be defined as the class of functions FO-reducible to a problem in NL). The other gives rise to the complexity class #L, consisting of functions that count the number of accepting computation paths of a non-deterministic logspace machine. As a consequence of the new upper bound on division and iterated multiplication, we get the following improvement of a result in [9]:

Corollary 6.9. *If* f *and* g *are in* #L, *and* f *is bounded by a polynomial in the length of its input, then* $\binom{f}{g}$ *is in* FNL.

It was observed in [2] that the techniques used here can be used to show that powering in small finite fields can be performed in FO. Other consequences of the new division algorithms are discussed in [3].

7. Small space-bounded complexity classes

For many people working in computational complexity theory, space-bounded computation only "begins" with logarithmic space. To be sure, there is a large literature dealing with space bounds between $\log\log n$ and $\log n$. (For example, see [39] for a perspective on the sequence of difficult papers leading up to a separation of the bounded-alternation hierarchy for sublogarithmic-space-bounded machines.) Nonetheless, this work relies on the automata-theoretic limitations of small-space-bounded machines. For instance, if $s(n) = o(\log n)$ is a fully-space-constructible function, then there is a constant k such that, for infinitely many n, $s(n) < k$. Thus every infinite unary language in $\mathrm{dspace}(o(\log n))$ has an infinite regular subset. This provides easy proofs of lower bounds for the space complexity of many languages, such as the proof in [30] that the set $\{0^n : n \text{ is prime}\}$ cannot be accepted in space $o(\log n)$.

However, it is still an open question whether the set of (binary encodings of) primes can be accepted in space $o(\log n)$. How can this be? Surely the binary encoding of a set cannot be easier than the unary encoding of the same set!

Let us see why this is still an open question. First, let us define some notation. Given any set A, the unary encoding of A, $\mathrm{un}(A)$ is the set $\{0^n : n \in A\}$, where we make use of the usual correspondence between natural numbers and binary strings.

Usually a lower bound on the complexity of the binary encoding of a set follows from a bound on the complexity of the unary encoding, using a standard *translation lemma*, such as:

Lemma 7.1 (Traditional translation lemma). *If* $s(\log n) = \Omega(\log\log n)$ *is fully space-constructible, then the first statement below implies the second*:

- $A \in \mathrm{dspace}(s(n))$.
- $\mathrm{un}(A) \in \mathrm{dspace}(\log n + s(\log n))$.

The converse also holds, if $s(\log n) = \Omega(\log n)$.

Note in particular that this translation lemma does not allow one to derive any lower bound on the space complexity of A, assuming only a logarithmic lower bound on the space complexity of $\mathrm{un}(A)$. As an example to see that this is unavoidable, consider the regular set $A = 10^*$. Arguing as in [30] it is easy to see that $\mathrm{un}(A) = \{0^{2^k} : k \in \mathbb{N}\}$ is not in $\mathrm{dspace}(o(\log n))$ (since it has no infinite regular subset).

There is another reasonable way to define space complexity classes. Let $\mathrm{DSPACE}(s(n))$ be the class of languages accepted by Turing machines that begin their computation with a worktape consisting of $s(n)$ cells (delimited by endmarkers), as opposed to the more common complexity classes $\mathrm{dspace}(s(n))$ where the worktape is initially blank, and the machine must use its own computational power to make sure that it respects the space bound of $s(n)$. Viewed another way, $\mathrm{DSPACE}(s(n))$ is simply $\mathrm{dspace}(s(n))$ augmented by a small amount of "advice", allowing the machine to compute the space bound. (This model was defined under the name "DEMON-SPACE" by Hartmanis and Ranjan [31]. See also Szepietowski's book [53] on sublogarithmic space. We have chosen to use the notation DSPACE and dspace

merely to let the capitalization emphasize that DSPACE has more computational power than dspace.)

DSPACE($s(n)$) seems at first glance to share many of the properties of dspace($s(n)$). In particular, it is still relatively straightforward to show that there are natural problems, such as the set of palindromes, that are not in DSPACE($o(\log n)$). (This follows from a simple crossing-sequence and Kolmogorov-complexity argument [31].)

The main contribution of this section is an easy argument, showing that the efficient division algorithm of Chiu et al. [19] provides a new translation lemma.

Lemma 7.2 (New translation lemma). *Let* $s(n) = \Omega(\log n)$ *be fully space-constructible. Then the following are equivalent*:

- $A \in \text{dspace}(s(n))$
- $\text{un}(A) \in \text{DSPACE}(\log\log n + s(\log n))$.

Proof. For the forward direction, it is sufficient to present a small-space algorithm for un(A).

Note that $\log\log n$ space can hold the binary representation of a short prime p. Thus on input 0^n, a DSPACE($\log\log n$) machine can compute the pieces of the Chinese Remainder Representation of n.

Thus, by [19] (see also Corollary 6.3), in space $\log(|n|) = \log\log n$ we can compute the bits of the binary representation of n. Hence, on input 0^n a Turing machine can simulate a $s(|n|)$-space-bounded computation (of a machine M having input n) in space $s(\log n)$, since $s(\log n)$ space is sufficient to store the worktape contents of the machine M, and it is also enough space to store the position of the input head of M (which requires only $\log|n| = \log\log n$ bits), as well as enough space to determine what symbol M is reading (i.e., the bits of the binary representation of n).

For the converse, given a Turing machine accepting un(A) in space $\log\log(x) + s(\log x)$ on input 0^x, we want to use $\log(|x|) + s(|x|) = O(s(|x|))$ space to determine if $x \in A$. We provide merely a sketch here.

The most naïve approach to carry out this simulation will not work, since we do not have enough space to record the location of the input head in a simulated computation on 0^x, and thus we cannot perform a step-by-step simulation. However, we do have enough space to carry out a simulation until either

(a) the input head returns to an endmarker without repeating a worktape configuration, or
(b) some worktape configuration is repeated.

In case (a), a step-by-step simulation is sufficient. In case (b), we can determine the period of the loop, and by means of some simple calculations that can be done in small space, we can determine the state the machine will be in when it encounters the other end marker.

Thus in either case, the simulation can proceed. □

Corollary 7.1. *Let* $\mathscr{C}$ *be any complexity class. In order to show* $\mathscr{C}$ *is not contained in* L, *it suffices to present a set* $A \in \mathscr{C}$ *such that* $\text{un}(A) \notin \text{DSPACE}(\log\log n)$.

We remark that the argument above can easily be adapted to show that the unary languages in NSPACE($\log\log n + \log(s(n))$) are exactly the unary encodings of languages in NSPACE($s(n)$). It should be remarked that a different translational method was presented by Szepietowski [52] for relating the L = NL question to the dspace($\log\log n$) = nspace($\log\log n$) question. However, as we have seen, there is no direct analog to Corollary 7.1 for the dspace or nspace classes.

In fact, it is not very difficult to show that there are unary languages in P (and even in dspace($(\log\log n)^2$)) that are not in DSPACE($\log\log n$). A straightforward delayed diagonalization (as in [30]) can be used to construct such a set $A \subseteq 0^*$. Note that this does not prove $P \neq L$, since un(A) (a very sparse set) *is* in DSPACE($\log\log n$). Stating this another way, the unary set $A \in \text{dspace}((\log\log n)^2)$ is equal to un(B) for some $B \in \text{dspace}((\log n)^2)$, where B is not known to be in P.

Observe that all unary languages in NSPACE($\log\log n$) are in FO. This follows since if B is a unary language in NSPACE($\log\log n$), then $B = \text{un}(A)$ for some $A \in \text{NL}$. Thus, by [43] (see also [29]), $A \in \text{RUD} = \bigcup_k \Sigma_k\text{TIME}(n)$. It was observed in [8] that $B = \text{un}(A) \in \text{FO}$ if and only if $A \in \text{RUD}$.

In some ways, DSPACE($\log\log n$) is a more natural class than dspace($\log\log n$), in the sense that this class is related to a natural class of branching programs, whereas no similar characterization is known for dspace($\log\log n$). The following result makes this more precise.

A branching program is *leveled* if the vertex set can be partitioned into *columns*, where all edges from vertices column i go to vertices in column $i+1$. We need *not* assume that all vertices in a given column query the same input location. We assume that vertices are labeled by a pair (c, j) where c is the number of the column, and j is the index of the node within column c. The *width* of a branching program is the maximum number of vertices in any column. In this paper, we consider only deterministic branching programs though parallel results on NSPACE classes and nondeterministic branching programs (or "contact schemes") can be obtained by the same techniques.

Theorem 7.1. *A is accepted by* DLOGTIME*-uniform branching programs of polynomial size and width $O(\log^{O(1)} n)$ if and only if A is* FO*-reducible to a language accepted by an oblivious* DSPACE($\log\log n$) *machine.*

Proof. First, consider a language accepted by an oblivious machine M with a worktape of size $O(\log\log n)$. By definition of "oblivious", the input location scanned by M at time t can be computed in FO. Thus it is an easy matter to construct a branching program with a node for each worktape configuration on each level, with edges simulating M's transition function. The resulting branching program will be FO-uniform, and this can be transformed into an equivalent DLOGTIME-uniform branching program by standard techniques.

Conversely, let A be accepted by a DLOGTIME-uniform leveled branching program of width $\log^{O(1)} n$. It is easy to show that there is a FO reduction that, given an input string x, produces a sequence of the form

$$\#\#f_1\#f_2\#\cdots\#f_t\#\#,$$

where t is the number of columns, and each f_i is a function $f_i : \{1, \ldots, w\} \to \{1, \ldots, w\}$, where $w = \log^{O(1)} n$ is the width of the branching program, with the property that $f_i(j) = j'$ iff the branching program, when in vertex j in column i, moves to vertex j' in column $i+1$ when querying the specified bit of x.

Note that an input x is accepted by M if and only if $f_t(f_{t-1}(\cdots(f_1(1))\cdots))$ is an accepting state of M. We encode each function f in the sequence as a list

$$(1, f(1))(2, f(2))\cdots(w, f(w)).$$

Note that there is an oblivious machine with space bound $O(\log\log n)$ that takes such a sequence of functions as input and computes the composition. □

Essentially, equivalent observations appear elsewhere. For instance, it is shown in [22] that leveled branching programs of width $O(2^{s(n)})$ correspond to non-uniform finite automata with space bound $s(n)$.

We do not know if the restriction to oblivious machines is necessary. If the behavior of a machine's input head is allowed to depend on the input contents, then the machine potentially has access to the $\log n$ bits of memory contained in the input head position. This might allow an otherwise space-bounded machine to solve L-complete problems. For example, the "non-uniform automata" of [11] are oblivious, correspond to constant-width poly-size branching programs and have the power of NC^1. But as shown by Barrington and Immerman (reported in [22]), if the obliviousness restriction is removed, the same machines have the power of general poly-size branching programs or L. But these machines make important use of non-uniformity, in the form of a read-only "program tape". It is not clear whether a DSPACE($\log \log n$) machine, for example, would be able to exploit the input position in the same way.

8. Conclusions

Our main theorem states that division and iterated multiplication are in fully uniform TC^0. This is significant on its own and also because it eliminates the most important example of a problem known to be in a circuit complexity class, but not known to be in the corresponding uniform complexity class.

We also proved that exponentiation modulo a number is in FO when the inputs have $O(\log n)$ bits. This result was quite unexpected, since the problem was previously not even known to be in FOM. It remains unknown whether exponentiation modulo a number with $\log^{O(1)} n$ bits is in FO, or even in FOM.

Finally, we have found a tight bound on the size of division and iterated multiplication problems that are in FO. We now know that these problems are in FO if and only if their inputs have $\log^{O(1)} n$ bits. Instances of the problems with larger inputs are known not to be in FO.

Acknowledgments

All three authors gratefully acknowledge the support of the NSF Computer and Computation Theory program. Much of this work was carried out during the March 2000 McGill Invitational Workshop on Complexity Theory—the authors thank the organizer Denis Thérien and all the other participants. They also thank Dieter van Melkebeek, Samir Datta, Michal Koucký, Rüdiger Reischuk, and Sambuddha Roy for helpful discussions.

Additional work on this project was carried out during the Park City Mathematics Institute's summer program in July and August 2000, supported by the Clay Mathematics Institute. The authors thank PCMI, CMI, Alexis Maciel, and the students in the PCMI undergraduate program where this material was presented.

References

[1] M. Agrawal, E. Allender, S. Datta, On TC^0, AC^0, and Arithmetic Circuits, J. Comput. System Sci. 60 (2000) 395–421.

[2] M. Agrawal, E. Allender, R. Impagliazzo, T. Pitassi, S. Rudich, Reducing the complexity of reductions, Comput. Complexity 10 (2001) 117–138.
[3] E. Allender, The division breakthroughs, L. Fortnow (Ed.), The Computational Complexity Column, EATCS Bull. 74, (2001), 61–77.
[4] E. Allender, A. Ambainis, D.A. Mix Barrington, S. Datta, H. LêThanh, Bounded depth arithmetic circuits: counting and closure, in: Proceedings of the 26th International Colloquium on Automata, Languages, and Programming (ICALP), Lecture Notes in Computer Science, Vol. 1644, 1999, pp. 149–158.
[5] E. Allender, D.A. Mix Barrington, Uniform circuits for division: consequences and problems, Electron. Colloq. Comput. Complexity 7 (2000) 65 (preliminary version of this paper).
[6] E. Allender, D.A. Mix Barrington, W. Hesse, Uniform circuits for division: consequences and problems, Proceedings of the 16th Annual IEEE Conference on Computational Complexity (CCC-2001), IEEE Computer Society Press, Silver Spring, MD, 2001, pp. 150–159.
[7] E. Allender, V. Gore, Rudimentary reductions revisited, Inform. Process. Lett. 40 (1991) 89–95.
[8] E. Allender, V. Gore, On strong separations from AC^0, in: Jin-Yi Cai (Ed.), Advances in Computational Complexity Theory, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Vol. 13, AMS Press, Providence, RI, 1993, pp. 21–37.
[9] E. Allender, K. Reinhardt, S. Zhou, Isolation, matching, and counting: uniform and nonuniform upper bounds, J. Comput. System Sci. 59 (1999) 164–181.
[10] A. Ambainis, D.A. Mix Barrington, H. LêThanh, On counting AC^0 circuits with negative constants, in: MFCS '98: Mathematical Foundations of Computer Science, Lecture Notes in Computer Science, Vol. 1450, Springer, Berlin, 1998, pp. 409–417.
[11] D.A. Mix Barrington, Bounded-width polynomial-size branching programs recognize exactly those languages in NC^1, J. Comput. System Sci. 38 (1989) 150–164.
[12] D.A. Mix Barrington, N. Immerman, Time, hardware, and uniformity, in: L.A. Hemaspaandra, A.L. Selman (Eds.), Complexity Theory Retrospective II, Springer, Berlin, 1997, pp. 1–22.
[13] D.A. Mix Barrington, N. Immerman, H. Straubing, On uniformity within NC^1, J. Comput. System Sci. 41 (1990) 274–306.
[14] D.A. Mix Barrington, P. Kadau, K.-J. Lange, P. McKenzie, On the complexity of some problems on groups given as multiplication tables, J. Comput. System Sci. 63 (2001) 186–200.
[15] P. Beame, S. Cook, J. Hoover, Log depth circuits for division and related problems, SIAM J. Comput. 15 (1986) 994–1003.
[16] H. Caussinus, P. McKenzie, D. Thérien, H. Vollmer, Nondeterministic NC^1 computation, J. Comput. System Sci. 57 (2) (1998) 200–212.
[17] A.K. Chandra, L. Stockmeyer, U. Vishkin, Constant depth reducibility, SIAM J. Comput. 13 (1984) 423–439.
[18] A. Chiu, Complexity of parallel arithmetic using the Chinese Remainder representation, Master's Thesis, University of Wisconsin-Milwaukee, 1995.
[19] A. Chiu, G. Davida, B. Litow, Division in logspace-uniform NC^1, RAIRO Theoret. Inform. Appl. 35 (2001) 259–276.
[20] P. Clote, Sequential machine independent characterizations of the parallel complexity classes AlogTIME, AC^k, NC^k, and NC, in: Feasible Mathematics: A Mathematical Sciences Institute Workshop held in Ithaca, New York, June 1989, Birkhauser, Basel, 1990, pp. 49–69.
[21] R. Constable, Type 2 computational complexity, in: Proceedings of the Fifth ACM Symposium on Theory of Computing (STOC), 1973, pp. 108–121.
[22] C. Damm, M. Holzer, Inductive counting for width-restricted branching programs, Inform. Comput. 130 (1996) 91–99.
[23] G. Davida, B. Litow, Fast parallel arithmetic via modular representation, SIAM J. Comput. 20 (1991) 756–765.
[24] L. Denenberg, Y. Gurevich, S. Shelah, Definability by constant-depth, polynomial-size circuits, Inform. Control 70 (1986) 216–240.
[25] P. Dietz, I. Macarie, J. Seiferas, Bits and relative order from residues, space efficiently, Inform. Process. Lett. 50 (1994) 123–127.
[26] W. Eberly, Very fast polynomial arithmetic, SIAM J. Comput. 18 (5) (1989) 955–976.
[27] W. Eberly, Logarithmic depth circuits for Hermite interpolation, J. Algorithms 16 (3) (1994) 335–360.
[28] R. Fagin, M.M. Klawe, N.J. Pippenger, L. Stockmeyer, Bounded-depth, polynomial-size circuits for symmetric functions, Theoret. Comput. Sci. 36 (2–3) (1985) 239–250.
[29] L. Fortnow, Time–space tradeoffs for satisfiability, J. Comput. System Sci. 60 (2000) 336–353.
[30] J. Hartmanis, L. Berman, On tape bounds for single letter alphabet language processing, Theoret. Comput. Sci. 3 (1976) 213–224.
[31] J. Hartmanis, D. Ranjan, Space bounded computations: review and new speculation, in: MFCS '89: Mathematical Foundations of Computer Science, Lecture Notes in Computer Science, Vol. 379, Springer, Berlin, 1989, pp. 49–66.

[32] J. Håstad, Almost optimal lower bounds for small depth circuits, in: S. Micali (Ed.), Randomness and Computation, Advances in Computing Research, Vol. 5, JAI Press, Greenwich, CT, 1989, pp. 143–170.
[33] J. Håstad, T. Leighton, Division in $O(\log n)$ depth using $n^{1+\varepsilon}$ processors, manuscript, available online at http://www.nada.kth.se/~johanh/papers.html.
[34] J. Håstad, I. Wegener, N. Wurm, S. Yi, Optimal depth, very small size circuits for symmetric functions in AC^0, Inform. Comput. 108 (2) (1994) 200–211.
[35] W. Hesse, Division is in Uniform TC^0, in: ICALP 2001: 28th International Colloquium on Automata, Languages and Programming, Lecture Notes in Computer Science, Vol. 2076, Springer, Berlin, 2001, pp. 104–114.
[36] N. Immerman, Descriptive Complexity, Springer, Berlin, 1999.
[37] N. Immerman, S. Landau, The complexity of iterated multiplication, Inform. Comput. 116 (1) (1995) 103–116.
[38] J. Johannsen, Weak bounded arithmetic, the Diffie–Hellman problem, and Constable's class K, in: Proceedings of the 14th IEEE Symposium on Logic in Computer Science (LICS), 1999, pp. 268–274.
[39] M. Liśkiewicz, R. Reischuk, Computing with sublogarithmic space, in: L.A. Hemaspaandra, A.L. Selman (Eds.), Complexity Theory Retrospective II, Springer, Berlin, 1997, pp. 197–224.
[40] I. Macarie, Space-efficient deterministic simulation of probabilistic automata, SIAM J. Comput. 27 (1998) 448–465.
[41] A. Maciel, D. Thérien, Threshold circuits of small majority-depth, Inform. Comput. 146 (1) (1998) 55–83.
[42] A. Maciel, D. Thérien, Efficient threshold circuits for power series, Inform. Comput. 152 (1) (1999) 62–73.
[43] V.A. Nepomnjaščiĭ, Rudimentary predicates and Turing calculations, Soviet Math. Dokl. 11 (1970) 1462–1465.
[44] I. Newman, P. Ragde, A. Wigderson, Perfect hashing, graph entropy, and circuit complexity (preliminary version), in: Proceedings of the Fifth Annual Structure in Complexity Theory Conference, IEEE Computer Society Press, Silver Spring, MD, 1990, pp. 91–99.
[45] J. Reif, Logarithmic depth circuits for algebraic functions, SIAM J. Comput. 15 (1986) 231–242.
[46] J. Reif, On threshold circuits and polynomial computation, in: Proceedings of the Second Annual Conference on Structure in Complexity Theory, IEEE Computer Society Press, Silver Spring, MD, 1987, pp. 118–123.
[47] J. Reif, S. Tate, On threshold circuits and polynomial computation, SIAM J. Comput. 21 (1992) 896–908.
[48] J. Reif, S. Tate, Optimal size integer division circuits, SIAM J. Comput. 19 (1990) 912–924.
[49] W.L. Ruzzo, On uniform circuit complexity, J. Comput. System Sci. 21 (1981) 365–383.
[50] N. Shankar, V. Ramachandran, Efficient parallel circuits and algorithms for division, Inform. Process. Lett. 29 (1998) 307–313.
[51] K. Siu, V.P. Roychowdhury, On optimal depth threshold circuits for multiplication and related problems, SIAM J. Discrete Math. 7 (1994) 284–292.
[52] A. Szepietowski, If deterministic and nondeterministic space complexities are equal for $\log\log n$, then they are also equal for $\log n$, Theoret. Comput. Sci. 74 (1990) 115–119.
[53] A. Szepietowski, Turing Machines with Sublogarithmic Space, in: Lecture Notes in Computer Science, Vol. 843, Springer, Berlin, 1994.
[54] H. Vollmer, Introduction to Circuit Complexity, Springer, Berlin, 1999.
[55] C.B. Wilson, Decomposing NC and AC, SIAM J. Comput. 19 (1990) 384–396.