

**COMPLEXITY OF PARALLEL ARITHMETIC  
USING THE CHINESE REMAINDER  
REPRESENTATION**

By  
Andrew Y. Chiu

A THESIS SUBMITTED IN  
PARTIAL FULFILLMENT OF THE  
REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

IN  
COMPUTER SCIENCE

at  
The University of Wisconsin-Milwaukee  
August 1995

**COMPLEXITY OF PARALLEL ARITHMETIC  
USING THE CHINESE REMAINDER  
REPRESENTATION**

By  
Andrew Y. Chiu

A THESIS SUBMITTED IN  
PARTIAL FULFILLMENT OF THE  
REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

IN  
COMPUTER SCIENCE

at  
The University of Wisconsin-Milwaukee  
August 1995

---

George Davida

Date

---

Graduate School Approval

Date

# COMPLEXITY OF PARALLEL ARITHMETIC USING THE CHINESE REMAINDER REPRESENTATION

By  
Andrew Y. Chiu

The University of Wisconsin-Milwaukee, 1995  
Under the Supervision of Professor George Davida

## ABSTRACT

In this thesis, we show that all of the fundamental arithmetic operations in the Chinese remainder representation can be performed in  $O(\log n)$  time using  $O(n^{O(1)})$  boolean processors. In particular, we present an  $O(\log n)$  depth,  $O(n^{O(1)})$  node, uniform circuit family for division, thereby, showing the division, iterated product, and powering problems to be in  $NC^1$ . We also present  $NC^1$  circuits for a wide variety of other arithmetic operations in the Chinese remainder representation, such as addition, multiplication, computation of the inverse, base extension, and conversion to and from binary.

**Keywords:** Parallel Arithmetic, Parallel Complexity, NC, Chinese Remainder Theorem, Number Representation.

---

George Davida

Date

## **Acknowledgements**

I would like to thank my family for their love and support throughout my struggles in writing my thesis, and generally throughout my school career. I would like to thank my advisor, Professor Davida, for encouraging me to make a second effort and then providing me guidance so that I may succeed. I would also like to thank Professor Peralta for his thorough reading of this thesis and his valuable suggestions.

# Contents

<b>Acknowledgements</b>	<b>v</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Background	1
1.2 A Brief Survey of Parallel Arithmetic Circuits	2
1.2.1 Addition	3
1.2.2 The Addition of $k$ Numbers and Multiplication	4
1.2.3 Division and The Multiplication of $k$ Numbers	6
1.3 Outline of the Thesis	7
<b>2 Chinese Remainder Representation and Circuit Complexity</b>	<b>9</b>
2.1 Chinese Remainder Representation	10
2.2 Circuits	13
2.3 Complexity Classes	14
2.4 $NC^1$ and Log-space	19
<b>3 Arithmetic operations in CRR</b>	<b>20</b>
3.1 Basic Arithmetic Operations	20
3.2 Converting from Binary to CRR	25
3.3 Rank and Other Related Problems	27
3.3.1 Computing Rank	27
3.3.2 The Mod Operation and Moduli Base Extension	32
3.3.3 Comparison	34
<b>4 Division</b>	<b>36</b>
4.1 Two Sub-problems of Division	36
4.1.1 CRR Scaling	36

4.1.2	CRR Shifting . . . . .	38
4.2	A P-Uniform Division Circuit . . . . .	39
4.3	Log-Space Uniform Division . . . . .	43
<b>5</b>	<b><math>U_E</math> Uniformity</b>	<b>49</b>
5.1	Uniform Circuits and Alternating Turing Machines . . . . .	49
5.2	$U_E$ Uniform Division . . . . .	55
<b>6</b>	<b>Conclusion</b>	<b>59</b>
6.1	Summary . . . . .	59
6.2	Further Research . . . . .	60

## List of Figures

1	Carry-look-ahead Addition . . . . .	4
2	Wallace tree for $k = 6$ . . . . .	5
3	Computation in a Wallace tree node . . . . .	5
4	A tree AND circuit for $n = 16$ . . . . .	15
5	When $\text{frac}[\sigma] < 3/4$ . . . . .	28
6	When $\text{frac}[\sigma] > 3/4$ . . . . .	29
7	A tree AND circuit for $n = 16$ . . . . .	50
8	A $\log n$ -ary tree AND circuit for $n = 16$ . . . . .	52
9	A circuit for computing $i^j \bmod m$ . Total time = $\frac{\log n}{\log \log n} \cdot \log \log n = \log n$ . . . . .	57

# Chapter 1

## Introduction

### 1.1 Background

Speeding up the fundamental arithmetic operations is of great importance in applications such as cryptography, computational geometry, and very high precision scientific work. Efforts to speed up the computations have included a number of approaches – from increasing the speed of single processors to the use of parallelism using “standard” computing elements. Parallel processing offers an attractive solution as it promises speed-up without requiring technological breakthroughs in high speed hardware design.

There is a fundamental theoretical issue in establishing the time required to compute division in parallel. With the exception of division, it has been known for some time that the other basic arithmetic operations, ie. addition and multiplication, can be performed in  $O(\log n)$  time using  $n^{O(1)}$  boolean processors. Using a well known geometric series [6], division can be computed in  $O(\log^2 n)$  time using  $n^{O(1)}$  processors. Thus, whether division has a higher parallel complexity than the other operations has been an open theoretical and practical problem.

If division could be computed within  $O(\log n)$  time, according to a common measure of parallel complexity known as  $NC^1$ , then the iterated product problem could be solved sequentially in logarithmic space [1]. Since the iterated product problem has been hypothesized as a candidate for a problem within PTIME - log-space, research into the parallel time complexity of division is also closely tied to low level space complexity.

Approaches to parallel division with better than  $O(\log^2 n)$  time, eg. [10, 1, 12, 3,

8], have all used alternate number representation systems, such as discrete Fourier transforms, or Chinese remaindering. These alternate systems represented numbers as small, independent units. This allowed for greater parallelism than ordinary binary arithmetic since the circuits were not limited by the carry.

Beame, Cook and Hoover [1] were the first to exhibit a  $O(\log n)$  time division circuit. However, a gap still remained between division and the other operations. The addition and multiplication circuits required minimal precomputation. In contrast, the division circuit required much precomputation which could not be performed within logarithmic time. It was uncertain whether the essence of division was computed within the circuit itself or during the lengthy precomputation stage. Consequently, the question of whether division had a higher parallel complexity than the other operations was still not totally resolved.

The main results in this thesis, Theorems 4.7 ( $U_B$  uniformity) and 5.6 ( $U_E$  uniformity), show that division can be computed in  $O(\log n)$  time by  $n^{O(1)}$  sized uniform circuits. Because our division circuits have logarithmic time restrictions on the precomputation stage, this thesis shows that division has the same parallel time complexity as multiplication, with both being in  $NC^1$ .

Our circuit uses the Chinese remainder representation to compute a geometric series approximation for the reciprocal quickly. Our approach is based on the  $O(\log n)$  depth division circuit presented in [1] and later extended by [3].

This thesis also presents  $O(\log n)$  time circuits for a wide variety of other arithmetic operations, such as addition, multiplication, and comparison, in the Chinese remainder representation.

## 1.2 A Brief Survey of Parallel Arithmetic Circuits

In this section, we describe the simplest, “fast” circuits for addition, multiplication and division. We only describe the basic ideas behind these circuits; the implementation details are beyond the scope of this work. This section provides important background material because efficient division depends on efficient multiplication,

and efficient addition. Also, this section illustrates some strategies involved in designing parallel arithmetic circuits. Finally, it provides the reader a baseline with which to compare with the more elaborate circuits described later in this thesis.

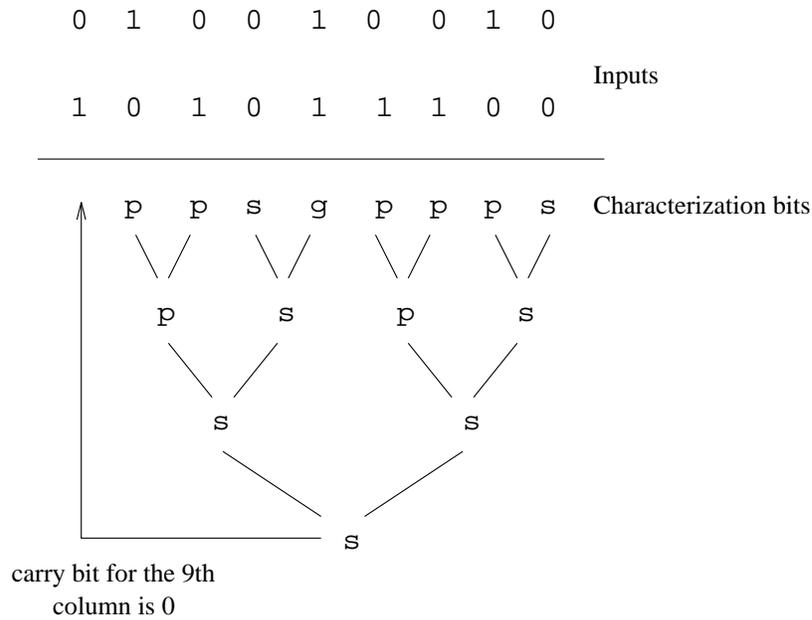
### 1.2.1 Addition

Addition is computable by  $O(\log n)$  depth circuits. The main idea behind these circuits is known as *carry-lookahead* [7, 5]. Carry-lookahead addition improves on the standard method of addition by computing the carry for every bit simultaneously. In the standard method, each carry bit is not computed until the computation in the previous column is done. Hence, a direct implementation of the standard algorithm produces an  $O(n)$  depth circuit. In contrast, a carry lookahead circuit compiles essential carry information and distributes them to all of the bits quickly using a tree circuit. Each carry bit is then computed simultaneously. Thus, carry-lookahead addition requires only  $O(\log n)$  depth.

In a carry-lookahead circuit, each column of inputs is first characterized as a  $g$  (generate),  $p$  (propagate) or  $s$  (stop) column. A column is characterized as  $g$  if both input bits are 1. This column will generate a carry when added. A column is characterized as  $p$  if one, and only one, of the input bits is a 1. This is because this column, when added, will propagate a carry from the previous column. Finally, a column is characterized as  $s$  if both input bits are 0. This column will not generate a carry even if the preceding column produces a carry. This characterization step requires only constant time when performed in parallel.

For each bit, we predict whether the previous column will produce a carry by looking at the characterization bits. The previous column will generate a carry if the first non- $p$  (non-propagating) bit to the right is a  $g$  bit (generating bit). Similarly, the previous column will not generate a carry if the first non- $p$  bit to the right is a  $s$  bit (stop bit). These bits are scanned quickly using a tree circuit, where each node takes the left bit if it is a  $g$  or an  $s$  (ie. if it generates or stops a carry), but takes the right bit if the left bit is a  $p$ . This is illustrated in figure 1. With a little care, the same tree can be reused to compute all of the carry bits. For a more detailed discussion, see [5, 7].

This scanning step requires  $O(\log n)$  depth to predict all of the carry bits. After we



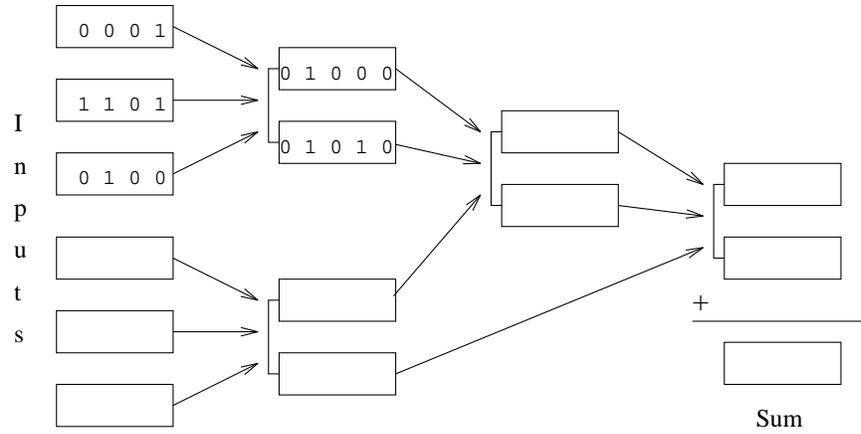


Figure 2: Wallace tree for  $k = 6$ .

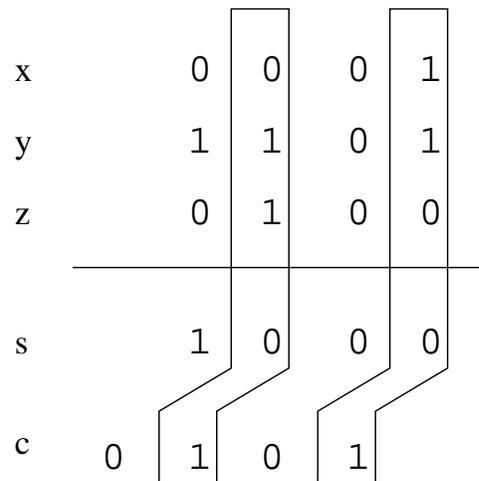


Figure 3: Computation in a Wallace tree node

a total of  $O(\log^2 n)$  time. The Wallace tree is faster because it avoids distributing the carry across the input bits until the very last stage. This illustrates an obvious, but important strategy in designing parallel arithmetic circuits. The circuit should compute with independent parts for as long as possible and only distribute the carry at the end. In some cases, one may have to resort to an alternate number representation system. In this case, by representing the sum as two numbers instead of one, the circuit computes on the columns independently and does not distribute the carry across the columns until the addition at the end.

Given an  $O(\log n)$  time circuit for adding  $k$  numbers, one can directly implement the “grade-school” method of multiplication to produce an  $O(\log n)$  time circuit for multiplication.

### 1.2.3 Division and The Multiplication of $k$ Numbers

The methods for computing division in less than  $O(\log^2 n)$  time all use alternate number representation systems, such as Fourier transforms or Chinese remaindering [10, 1, 12, 8]. For purely binary division, the simplest  $O(\log^2 n)$  method is to compute the well known geometric series,

$$1/y = 1 + (1 - y) + \cdots + (1 - y)^i + \cdots, \text{ when } 0 < y < 1 \quad (1)$$

By scaling the divisor into the proper range, one can use the above approximation series to approximate the reciprocal, which can then be multiplied with the dividend to obtain the result. The problem of division thus reduces to calculating the terms,  $(1 - y)^i$ , in the series. In other words, division reduces to the problem of multiplying  $k$   $n$ -bit numbers, where  $k = O(n)$  [1].

The most obvious method for multiplying  $k$  numbers is to multiply them pairwise in parallel, forming a binary tree. Since each multiplication requires  $O(\log n)$  time and there are a total of  $O(\log n)$  levels, this produces a circuit with  $O(\log^2 n)$  depth. As we saw in our previous discussion, however, this method may not produce the optimal solution for the problem. One way to improve upon this result is to use an alternate number representation system for the intermediate results so that one avoids dealing with the carry until the final step. This alternative representation should allow multiplication with small independent parts. The two most common candidates for this is the Fourier transform and the Chinese remaindering representation.

By using the Fourier transform, a circuit can compute division in  $O(\log n \log \log n)$  depth [10]. The Fourier transform division circuit is quite complicated and we will not describe the details here. The basic idea behind that circuit is that one can multiply  $O(\log n)$  numbers in  $O(\log n)$  time using the Fourier transform. Therefore, a circuit can compute the above approximation series in  $O(\log n \log \log n)$  time by applying the Fourier transform recursively. For a full description, see [10].

By using the Chinese remainder representation (CRR), a circuit can compute division in  $O(\log n)$  depth [1]. One problem with CRR is that most CRR arithmetic circuits, including the division circuit in [1], require a lot of precomputation. This causes doubt about the true complexity of a CRR circuit because it is difficult to determine whether the bulk of the computation is done within the circuit itself, or during the precomputation stages. To remedy this, a criteria known as *uniformity* was developed to restrict precomputation so that the depth of the circuit is the true measure of the problem's time complexity. In the rest of this thesis, we will present a division circuit, based on CRR, with  $O(\log n)$  depth and show that our circuit is uniform. Thus, we will show that division can truly be computed in  $O(\log n)$  time.

### 1.3 Outline of the Thesis

In chapter 2, we briefly summarize the theoretical foundations of our result. We describe the Chinese remainder representation and some of its important properties. We also describe the circuit complexity model and the complexity class known as  $NC$ .

In chapter 3, we present  $NC^1$  circuit families for basic arithmetic operations in CRR. In section 3.1, we present circuits to compute addition, multiplication, addition of  $O(n)$  numbers and multiplication of  $O(n)$  numbers. In section 3.2, we present circuits to convert a binary number to a CRR number. Finally, in section 3.3, we present circuits to compute the mod operation, comparison and other related problems. These circuits form a “toolbox” from which to build our  $NC^1$  division circuit.

In chapter 4, we present our  $NC^1$  circuit for division. We start off, in section 4.1, by describing circuits for two restricted cases of the division problem — CRR scaling and CRR shifting. In section 4.2, we describe the circuit given in [1]. As mentioned above, the circuit in [1] is the first circuit with  $O(\log n)$  depth and is the basis for our

own division circuit. Therefore, we describe the circuit in detail in order to present the reader with background material as well as a baseline for comparison. Finally, we present our  $NC^1$  division circuit in section 4.3. Our division circuit computes a geometric series approximation. It has  $O(\log n)$  depth because of the fast, parallel arithmetic possible in CRR.

In chapter 5, we show that our division circuit is in  $NC^1$  even under the more restricted  $U_E$  definition of uniformity.

In chapter 6, we summarize our result and point out further avenues of research.

## Chapter 2

# Chinese Remainder Representation and Circuit Complexity

In this chapter, we briefly summarize the theoretical foundations of our result. First, we describe the Chinese remainder representation system (CRR). The main benefit of CRR is that each number is represented by many small, independent parts. This allows for greater opportunities to parallelize since we do not have to worry about a “carry” as in ordinary binary arithmetic. The main disadvantage of CRR is that it is cyclic. Hence, there is no easy way to determine the magnitude of a number. We will remedy this by introducing the concept of *rank*.

We then describe the circuit complexity model. The circuit model has emerged as an important model of parallel complexity since it can be easily implemented, and it provides a reasonable measure of the parallel hardware size and parallel execution time. The main difference between the circuit model and the more traditional algorithmic model is that a circuit only solves a problem for inputs of a fixed size. Hence, a complete problem solution requires a family of circuits for different input sizes. This motivates the concept of *uniformity*, which requires that there be a simple rule for generating the description of the circuit for any given size. Uniformity is critical to the circuit model because it ensures that the bulk of the computation is done within the circuit rather than during the circuit generation stage. Therefore, it provides a fuller picture of circuit complexity than the one just considering parallel execution time and hardware size.

Finally, we describe the parallel complexity class  $NC$  and present some theorems linking it to division and the sequential complexity classes.

## 2.1 Chinese Remainder Representation

**Definition 2.1** Let  $\mathcal{M} = \langle m_{r-1}, \dots, m_0 \rangle$  be a set of pairwise relatively prime numbers with product  $M$ . An integer  $0 \leq x < M$  can be represented in the Chinese remainder representation as the vector  $[x_{r-1}, \dots, x_0]$  where

$$x_i = x \bmod m_i$$

We call a CRR system an  $n$ -bit CRR system if  $\lfloor \log M \rfloor = n$ .<sup>1</sup> This  $n$ -bit system is written as  $\text{CRR}(\mathcal{M})$ . The vector  $\mathcal{M} = \langle m_{r-1}, \dots, m_0 \rangle$  is called the moduli base of our  $n$ -bit CRR. Integers represented in  $n$ -bit CRR systems are called  $n$ -bit CRR integers, and are written as  $[x]_{\mathcal{M}}$ .

In this thesis, we will only use primes  $p > 3$ , and  $2^j$ , where  $j = O(\log n)$ , in our moduli base. The index,  $j$ , is chosen such that  $2^j$  is larger than the largest prime in our moduli base. Limiting our moduli to primes and  $2^j$ 's allows us to take advantage of the good number theoretic properties of these moduli, namely the easy calculation of inverses and the existence of generators.

Also, we will write all CRR integers with square brackets,  $[x_{r-1}, \dots, x_0]$  and all the moduli base vectors with angle brackets,  $\langle m_{r-1}, \dots, m_0 \rangle$ . We will understand  $|x^{-1}|_m$  to denote the modular inverse of  $x \bmod m$ . Finally, we shall interchangeably write  $|x|_m$  for  $x \bmod m$ .

Given an arbitrary CRR integer  $[x_{r-1}, \dots, x_0]$ , we derive its radix value  $x$  via the Chinese remainder theorem [9].

### Theorem 2.1 (Chinese remainder theorem)

$$x = \left( \sum_{i < r} x_i \nu_i \right) \bmod M, \quad (2)$$

where,

$$\nu_i = \frac{M}{m_i} \cdot \left| \left( \frac{M}{m_i} \right)^{-1} \right|_{m_i} \quad (3)$$

From the properties of modular arithmetic [9], we see that this system allows “carry-free” arithmetic. Specifically, let  $x = [x_{r-1}, \dots, x_0]$  and  $y = [y_{r-1}, \dots, y_0]$ . We

---

<sup>1</sup>In this thesis, all log operations are on base 2 unless otherwise specified.

have,

$$x + y = [|x_{r-1} + y_{r-1}|_{m_{r-1}}, \dots, |x_0 + y_0|_{m_0}] \text{ for } x + y < M, \text{ and} \quad (4)$$

$$x \cdot y = [|x_{r-1}y_{r-1}|_{m_{r-1}}, \dots, |x_0y_0|_{m_0}] \text{ for } xy < M. \quad (5)$$

**Example 2.1 :** Let  $\mathcal{M} = \langle 8, 7, 5 \rangle$  with  $M = 280$ . Note that  $\mathcal{M}$  is a sufficient moduli base for an 8-bit CRR. Let  $x = [3, 2, 1]$  and  $y = [5, 5, 0]$ . We calculate the basis vectors from the Chinese remainder theorem as,

$$\nu_2 = (280/8)|(280/8)^{-1}|_8 = 35|3^{-1}|_8 = 105$$

$$\nu_1 = (280/7)|(280/7)^{-1}|_7 = 40|5^{-1}|_7 = 120$$

$$\nu_0 = (280/5)|(280/5)^{-1}|_5 = 56|1^{-1}|_5 = 56$$

Applying the Chinese remainder theorem, we get

$$x = 3(105) + 2(120) + 1(56) \bmod 280 = 611 \bmod 280 = 51$$

$$y = 5(105) + 5(120) + 0(56) \bmod 280 = 1125 \bmod 280 = 5$$

Checking (4) and (5), we see that indeed,

$$x + y = [0, 0, 1] = 0(105) + 0(120) + 1(56) \bmod 280 = 56$$

$$xy = [7, 3, 0] = 7(105) + 3(120) + 0(56) \bmod 280 = 255$$

The actual efficiency of CRR addition and multiplication depends on the size of each  $m_i$ . If each  $m_i$  is an  $O(n)$  bit integer, then the arithmetic is actually less efficient since we have to compute mod  $M$  after every operation. Fortunately, the prime number theorem allows us to show that each  $m_i$  need only be an  $O(\log n)$  bit integer and we only need  $O(n/\log n)$  bit moduli in an  $n$ -bit CRR. The next lemma verifies this claim.

**Lemma 2.2** *For any positive integer  $n$  there is a CRR consisting of  $O(\log n)$  bit moduli capable of uniquely representing every integer below  $2^n$ .*

**Proof :** Let  $\pi(n)$  denote the number of primes below  $n$ . By the prime number theorem, [9], we know that

$$c \frac{n}{\log n} \leq \pi(n) \leq d \frac{n}{\log n},$$

for some constants  $c$  and  $d$ . To see that there are enough primes with  $O(\log n)$  bits to represent  $x \leq 2^n$ , consider the  $O(\log n)$  bit primes  $p_1, \dots, p_r$  between  $2^{l-\delta}$  and  $2^{l+\delta}$ , where  $l = \lceil \log n \rceil$  and  $\delta$  some positive constant. Let  $P$  denote their product  $p_1 \cdots p_r$ . Then,

$$\begin{aligned} \log P &= (\text{total number of primes}) \cdot (\text{number of bits in each prime}) \\ &\geq \left(c \frac{2^{l+\delta}}{l+\delta} - d \frac{2^{l-\delta}}{l-\delta}\right)(l-\delta) \\ &\geq 2^{l-\delta} \left(c 2^{2\delta} \frac{l-\delta}{l+\delta} - d\right) \\ &\geq n, \end{aligned}$$

for some constant  $\delta$  satisfying

$$c 2^{2\delta} \frac{l-\delta}{l+\delta} - d \geq 2^\delta$$

Therefore,  $O(\log n)$  bits are sufficient for each moduli in an  $n$ -bit CRR.  $\square$

Finally, we define *rank*. We re-write the Chinese remainder theorem as follows [3]:

$$\sum_{i < r} x_i v_i = \mathcal{R}(x, \mathcal{M})M + x \quad (6)$$

**Definition 2.2** *The integer  $\mathcal{R}(x, \mathcal{M})$  in equation (6) is called the rank of  $x$  with respect to the moduli base  $\mathcal{M}$ .*

CRR is a modular arithmetic system and so is cyclic. In a strict CRR system, there is no notion of the magnitude of a number. This makes operations such as comparison and division much more difficult [14, 8, 4]. Computing the rank  $\mathcal{R}(x, \mathcal{M})$  allows us to counter the “wraparound” effect when applying the Chinese remainder theorem. For example, rank was used to solve comparison in CRR in [3]. Unfortunately, computing rank is not a trivial operation. We will present a “fast” circuit for computing rank in section 3.3.1 after we formally describe our circuit model and develop some circuits for basic CRR arithmetic.

**Example 2.2 :** Let  $\mathcal{M} = \langle 8, 7, 5 \rangle$  with  $M = 280$ . Let  $x = [3, 2, 1]$ . By the Chinese remainder theorem,

$$x = 3(105) + 2(120) + 1(56) \bmod 280 = 611 \bmod 280 = 51.$$

Following equation (6), we rewrite the above as,

$$\begin{aligned}x + \mathcal{R}(x, \mathcal{M}) &= 611 \\51 + \mathcal{R}(x, \mathcal{M})280 &= 611 \\ \mathcal{R}(x, \mathcal{M}) &= 2\end{aligned}$$

Therefore, the rank of  $[3, 2, 1]$  with respect to the moduli base  $\mathcal{M}$  is 2.

## 2.2 Circuits

**Definition 2.3** *A Boolean circuit is a labeled acyclic, directed graph (DAG). Nodes are labeled as input, constant, AND, OR, NOT, or output nodes. Input and constant nodes have zero fan-in. Output nodes have fanout zero.*

*A circuit with  $n$  input nodes and  $m$  output nodes computes a boolean function  $f : \{0, 1\}^n \rightarrow \{0, 1\}^m$ . The size of a circuit is the number of edges and the depth of a circuit is the longest path from some input node to some output node.*

Informally, a *problem* can be specified by a *transducer of strings*, ie. a function that associates a set of binary output strings with the set of binary input strings. Given a *family of circuits*,  $C = \{C_i\}$ , for  $i = 1, 2, \dots$ , we say  $C$  solves a problem  $P$  if the function computed by  $C_i$  matches precisely the string transduction required by  $P$  for inputs of length  $i$ .

Here lies the fundamental difference between the algorithmic model and the circuit model. An algorithm that computes a particular problem typically solves that problem for all input sizes, assuming that there are enough resources. In the circuit model, we have a different (and perhaps radically different) circuit for each input size. This motivates the concept of *uniformity*. Informally, a uniform family of circuits for a problem is one where there is a simple rule for constructing the circuit for each input size. This allows us to more accurately classify the parallel complexity of a problem in the circuit model, because we restrict the bulk of the computation within the circuit itself, rather than during the construction of the circuit.

For instance, suppose we had a problem  $P$  whose output depended solely on the size of the input. We can then construct a non-uniform circuit family for  $P$ , which requires only constant depth for any size input. During the construction of a circuit

$C_i$  for an input size  $i$ , we simply compute the required output and hardwire it into the circuit with constant nodes. In this way, an arbitrarily difficult problem can be classified in the non-uniform circuit model as a very simple problem, requiring constant depth.

## 2.3 Complexity Classes

We first formalize our notion of uniformity. We measure the difficulty of “constructing” a circuit by the amount of computation needed to produce a description of that circuit. Since a circuit is a labeled DAG, an adjacency list is a natural way to describe it. The following definition is from [2].

**Definition 2.4** *A family of circuits  $C$  is log-space uniform if the description of the  $n$ -th circuit  $C_n$  can be generated by a Turing machine using only  $O(\log n)$  workspace with a length  $n$  unary input string.*

**Definition 2.5** *The class  $NC^k$  consists of functions which are computable by a log space uniform family of circuits with  $O(\log^k n)$  depth and polynomial size. The class  $NC = \bigcup_{k \geq 1} NC^k$ .*

From the definition, one sees that the class  $NC$  consists of functions which are both amenable to parallelization, *and*, feasible for parallelization. They are amenable to parallelization because the potential speedup is large ( $\approx poly(n)/polylog(n)$ ). They are feasible to parallelize because the hardware costs are small, only a polynomial in the input size. A large number of problems from a wide variety of areas, such as sorting, graph theory, and arithmetic, have been shown to be in  $NC$  [7, 6, 10]. This class is also robust under many other models of parallel computation, like PRAMS [2, 6, 11].

Our Turing machine model is an “off-line” machine with a two-way read only input tape. We have a separate work tape, the size of which is bounded by the definition above. Normally, a Turing machine is considered an acceptor of inputs; in our case, we are transducing a circuit, so we also need a write only output tape in the machine model.

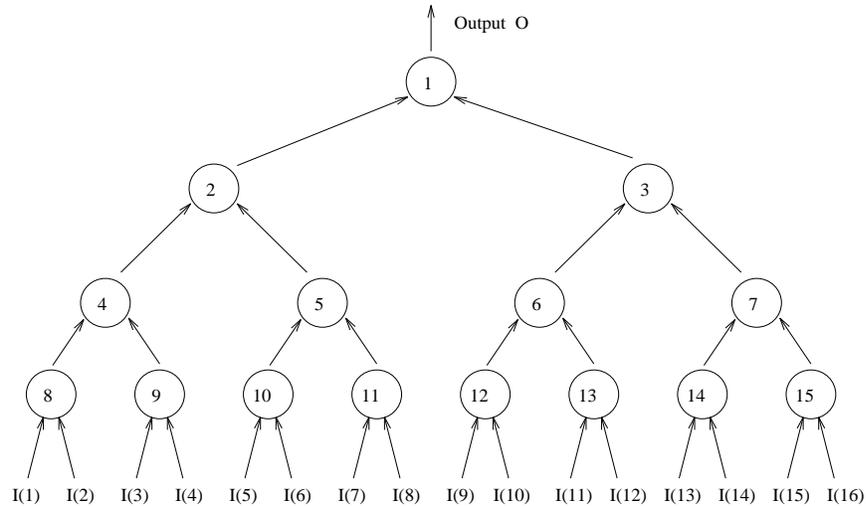


Figure 4: A tree AND circuit for  $n = 16$ .

Although the formal definition of log space uniformity seems rather esoteric, in practice, it is relatively simple and surprisingly inclusive. Most commonly used subcircuits are log-space uniform, like tables, selectors, and small arithmetic units. Furthermore, we can combine these uniform subcircuits into arrays and trees and still maintain log-space uniformity. For most circuits, the real problem in showing uniformity lies in showing that the precomputed constants can be generated within log-space limitations.

**Example 2.5 :** Consider the simple tree circuit which ANDs together  $n$  binary values. We will show that this circuit is log-space uniform, ie. it can be generated by a Turing machine in  $O(\log n)$  space. Because it has  $O(\log n)$  depth, this tree circuit is in  $NC$ .

First, we label the nodes. We label the input nodes as  $I(1), \dots, I(n)$  and the output node as  $O$ . We label the root node as 1. If a node is labeled  $i$ , then we label its left child as  $2i$  and its right child as  $2i + 1$ . This labeling scheme is shown in figure 4. We can then generate the tree circuit for any  $n$  with the following program.

```

generate(n)
{
    int i, parent, left, right;

    /* Output the root node's connections */

```

```

output(1, TYPE, AND);
output(1, OUTPUT, 0);
output(1, LEFT_INPUT, 2);
output(1, RIGHT_INPUT, 3);

/* Output the internal node's connections */
for (i=2;i<n/2; i++) {
    output(i, TYPE, AND);
    parent = i/2; left = 2*i; right=left+1;
    output(i, OUTPUT, parent);
    output(i, LEFT_INPUT, left);
    output(i, RIGHT_INPUT, right);
}

/* Output the last layer's connections */
for (i=n/2-1; i<n-1; i++) {
    output(i, TYPE, AND);
    parent = i/2; left = 2*i-n+1; right=left+1;
    output(i, OUTPUT, parent);
    output(i, LEFT_INPUT, I(left));
    output(i, RIGHT_INPUT, I(right));
}
}

```

The above program runs in  $O(\log n)$  space because it only uses five variables,  $n$ ,  $i$ ,  $parent$ ,  $left$ , and  $right$ , which each require only  $O(\log n)$  bits. Note that our program is allowed even though it runs in  $O(n)$  time. This is because for log-space uniformity, we have a logarithmic bound on space only. Our time bound is polynomial. This helps explain the expressive power of a log-space Turing machine. Because our time bounds are so loose, our programs can run for a very long (polynomial) time in order to generate the circuit description. The only requirement is that it reuses the same logarithmic space over and over again during execution. With this intuition, one sees how a large variety of circuit types, such as trees, arrays, selectors, etc., can be generated within logspace limitations.

Now, suppose that the nodes in our tree are not AND nodes, but black boxes containing some  $NC^1$  circuit. Let each blackbox take two inputs and produce one output. Furthermore, let the size of each black box be `node_size`, and let its description be generated by `generate_node(start)`, where `start` represents its input node, `start+node_size-1` represents its output node, and `start ... , start + node_size - 1` denote the label numbers which the black box may use. Now, we can generate the combined tree circuit with the following program.

```
#define NEW_START(i) ((i-1)*node_size+1)
#define NEW_OUTPUT(i) (i*node_size)
#define NEW_INPUT(i) ((i-1)*node_size+1)

generate(n)
{
    int i, parent, left, right;

    /* Output the root node's connections */
    generate_node(NEW_START(1));
    output(NEW_OUTPUT(1), OUTPUT, 0);
    output(NEW_INPUT(1), LEFT_INPUT, NEW_OUTPUT(2));
    output(NEW_INPUT(1), RIGHT_INPUT, NEW_OUTPUT(3));

    /* Output the internal node's connections */
    for (i=2;i<n/2; i++) {
        generate_node(NEW_START(i));
        parent = i/2; left = 2*i; right=left+1;
        output(NEW_OUTPUT(i), OUTPUT, NEW_INPUT(parent));
        output(NEW_INPUT(i), LEFT_INPUT, NEW_OUTPUT(left));
        output(NEW_INPUT(i), RIGHT_INPUT, NEW_OUTPUT(right));
    }

    /* Output the last layer's connections */
    for (i=n/2-1; i<n-1; i++) {
```

```

    generate_node(NEW_START(i));
    parent = i/2; left = 2*i-n+1; right=left+1;
    output(NEW_OUTPUT(i), OUTPUT, NEW_INPUT(parent));
    output(NEW_INPUT(i), LEFT_INPUT, I(left));
    output(NEW_INPUT(i), RIGHT_INPUT, I(right));
}
}

```

This program is substantially the same as the previous one, except that this program uses `generate_node` to produce the detailed description of each node, and uses `NEW_START`, `NEW_INPUT`, and `NEW_OUTPUT` to transform the node labels from the old labeling system to the new one. Since each node is presumed to be in  $NC^1$  and our program only uses five additional variables, the whole program runs in log-space and polynomial time. Thus, the combined circuit is also in  $NC^1$ . This further illustrates the expressive power of a log-space Turing machine. One can combine basic  $NC^1$  circuits into larger tree or array structures and still maintain log-space uniformity. This way, one sees how the structure of most common circuits are log-space uniform (since they are often just combinations of smaller log-space uniform circuits). As mentioned above, the real problem in showing uniformity lies in showing that all the precomputed constants can be generated in log-space.

For the subclass of  $NC^1$ , there is a newer and stricter definition of log-space uniformity known as  $U_E$  uniformity (the previous uniformity is known as  $U_B$  uniformity) [11]. For  $U_B$  uniformity, we are concerned about the complexity of generating the global circuit layout. For  $U_E$  uniformity, we are concerned only with computing the local connections from any node.  $U_E$  is stricter than  $U_B$  uniformity since in  $U_E$  uniformity, we are limited in both time and work space for computing the local connections, while in  $U_B$  uniformity, we are only limited in work space.

In this thesis, we will first work with  $U_B$  uniformity because our results are based on the approach of [1]. The approach of [1] is based on  $U_B$  uniformity and it is not immediately obvious how to extend it to  $U_E$  uniformity. In particular, important discrete log tables seem difficult to generate under  $U_E$  uniformity. Therefore, for a clearer discussion, we will first show that our  $O(\log n)$  division circuit family is  $U_B$  uniform. In section 5, after we have shown  $U_B$  uniformity, we will formally describe

the  $U_E$  uniformity model and extend our result to show that our division circuit family is also  $U_E$  uniform.

## 2.4 $NC^1$ and Log-space

We collect two facts relating circuit complexity and space complexity. The first fact is a specialization of Theorem 4 in [2]. Let *log-space* represent the class of problems which can be solved on a sequential Turing machine with  $O(\log n)$  space and polynomial time [2], for an input of size  $n$ .

**Theorem 2.3**  $NC^1 \subseteq \text{log-space}$ .

The second fact is established by Theorem 6.2 in [1]. Let the *iterated product* problem be the computation of the product of  $n$ ,  $n$ -bit binary numbers. Let the *powering* problem be the special case of the iterated product problem in which all the factors are the same.

**Theorem 2.4** *If division, iterated product, or powering is in  $NC^1$ , then they are all in  $NC^1$ .*

## Chapter 3

### Arithmetic operations in CRR

Before we present our division circuit, we need some circuits for basic arithmetic operations in CRR. In this chapter we present  $NC^1$  circuit families for

- basic arithmetic operations in CRR, such as addition, addition of  $O(n)$  integers, multiplication, multiplication of  $O(n)$  integers, and computation of inverses,
- converting from binary to CRR, and
- rank and other related problems, such as the mod operation, moduli base extension and comparison.

#### 3.1 Basic Arithmetic Operations

**Theorem 3.1** *Let  $x, y, z_1, \dots, z_k$ , be  $n$ -bit CRR integers with a common moduli base  $\mathcal{M} = \langle m_{r-1}, \dots, m_0 \rangle$ , where  $k = O(n)$ , each  $m_i$  is an  $O(\log n)$  bit integer and  $r = O(n/\log n)$ . Let  $M$  denote the product  $m_{r-1} \cdots m_0$ . We can compute*

1.  $A_1 = (x + y) \bmod M$ ,
2.  $A_2 = (xy) \bmod M$ ,
3.  $A_3 = (z_1 + z_2 + \cdots + z_k) \bmod M$ ,

with  $NC^1$  circuit families.

**Proof :** Let  $x = [x_{r-1}, \dots, x_0]$ ,  $y = [y_{r-1}, \dots, y_0]$ ,  $A_1 = [A_{(1,r-1)}, \dots, A_{(1,0)}]$  and  $A_2 = [A_{(2,r-1)}, \dots, A_{(2,0)}]$ . Then,

$$A_{(1,j)} = x_j + y_j \bmod m_j \quad (7)$$

$$A_{(2,j)} = x_j \cdot y_j \bmod m_j \quad (8)$$

We compute  $A_1$  with an  $NC^1$  circuit family which computes each  $A_{(1,j)}$  in parallel via (7). This circuit family is obviously log-space uniform and has  $O(\log n)$  depth, since  $x_j, y_j, m_j$  are all  $O(\log n)$ -bit integers. Similarly, we have an  $NC^1$  circuit family for computing  $A_2$  via (8).

Let each  $z_i = [z_{(i,r-1)}, \dots, z_{(i,0)}]$ ,  $A_3 = [A_{(3,r-1)}, \dots, A_{(3,0)}]$ . Then,

$$A_{(3,j)} = \sum_{i=1}^k z_{(i,j)} \bmod m_j. \quad (9)$$

$A_3$  is computed with an  $NC^1$  circuit family via (9). (9) is an addition of  $k$   $O(\log n)$ -bit binary integers, which can be computed in  $O(\log n)$  depth using the Wallace tree circuit described in section 1.2.2.  $\square$

We would also like to compute the multiplication of  $k$   $n$ -bit CRR integers with  $NC^1$  circuit families. From our discussion above, we see that this problem reduces to the problem of computing the multiplication of  $k$   $O(\log n)$  bit integers mod  $m_j$ . Unfortunately, there is no straight forward method for doing this in  $O(\log n)$  depth in binary. We shall use an alternate method, given in [1], which takes advantage of the number theoretic properties of our moduli  $m_j$ . We first recall two facts from number theory [9].

**Lemma 3.2** *Given a prime  $p$ , there exists a generator or primitive element  $g$ , such that every number  $0 < w < p$  can be written as*

$$w = g^l \bmod p.$$

where  $0 \leq l < p$ .  $l$  is known as the discrete log of  $w$ .

**Lemma 3.3** *Every integer  $0 < w < 2^r$ ,  $r \geq 2$ , can be written as*

$$w = (-1)^{l_1} \cdot 2^{l_2} \cdot 5^{l_3} \bmod 2^r,$$

where,  $l_1 = 0$  or  $1$ ,  $l_2 < r$  and  $0 \leq l_3 < 2^{r-2}$ .

These two lemmas state that every number in our product (remember that our moduli are restricted to be either a prime  $p > 3$ , or of the form  $2^j$ ) can be transformed into an index. Hence, the multiplication of  $k$  numbers can be rewritten as an addition of  $k$  indices. This idea forms the basis for our next theorem.

**Theorem 3.4** *Let  $m, w_1, \dots, w_k$ , where  $k = O(n)$ , be  $O(\log n)$  bit integers. Let  $m$  be either an  $O(\log n)$  bit prime  $p > 3$  or of the form  $2^j$ , where  $j = O(\log n)$ . We can compute  $W = (w_1 \cdots w_k) \bmod m$  with an  $NC^1$  circuit family.*

**Proof :** If  $m$  is prime, we perform the multiplication  $W = (w_1 \cdots w_k) \bmod m$ , using the following  $NC^1$  circuit family. Let  $T_p$  be a table of powers, and  $T_d$  be a table of discrete logs  $\bmod m$ .

1. Check if any  $|w_i|_m = 0$ . If so, return  $W = 0$ .
2. In parallel, look up in the discrete log tables,  $l_i = T_d(|w_i|_m, m)$ . Table lookup can be done in  $O(\log n)$  steps since  $T_d$  has only  $O(\text{poly}(n))$  entries.
3. Compute  $L = \sum_{i=1}^k l_i \bmod (m - 1)$ . This is just a sum of  $k$  numbers which can be done in  $NC^1$ .
4. Look up in the table of powers,  $W = T_p(L, m)$ .

This circuit works by converting the  $w_i$ 's to indices, adding up the indices and then powering to obtain the result. It has a depth of  $O(\log n)$  since we can do table lookup and add  $k$  numbers in  $O(\log n)$  depth. The circuit family is log-space uniform since the tables  $T_d$  and  $T_p$  can be generated by brute force in log-space. For instance, we can generate  $T_d(x_i, m)$  by

1. Find the generator  $g$  for a moduli  $m$ . For each  $i < m$ , check every power,  $l \leq m - 1$ , of  $i$ . If some  $i^l = 1$  then  $i$  is not the generator.
2. For each entry  $x_i$ , check every power,  $l \leq m - 1$ , of  $g$  until  $g^l = x_i$ .

Similarly,  $T_p$  can be generated in log-space. The above algorithm runs in polynomial time and log-space, since all  $x_i, g, m$  are all  $O(\log n)$  bit integers. Once again, this is allowed because log-space uniformity places logarithmic bounds on space only. The program can run for a very long (polynomial) time as long as it stays within its logarithmic workspace. Therefore, if  $m$  is a prime  $p > 3$ , we can compute  $W = (w_1 \cdots w_k) \bmod m$  with an  $NC^1$  circuit family.

If  $m$  is of the form  $2^r$ , we use the same idea as above except we have three discrete log values and three additions instead of one.

1. Check if any  $|w_i|_m = 0$ . If so, return  $W = 0$ .
2. Look up in the discrete log tables,  $(l_{1i}, l_{2i}, l_{3i}) = T_d(|w_i|_m, m)$ .
3. Compute  $(L_1, L_2, L_3)$ , where

$$L_1 = \sum_{i=1}^k l_{1i} \bmod 2 \quad (10)$$

$$L_2 = \sum_{i=1}^k l_{2i} \quad (11)$$

$$L_3 = \sum_{i=1}^k l_{3i} \bmod 2^{r-2} \quad (12)$$

4. Look up in the table of powers,  $(W_1, W_2, W_3) = (T_p(L_1, m), T_p(L_2, m), T_p(L_3, m))$ .
5. Compute  $W = W_1 \cdot W_2 \cdot W_3 \bmod m$ .

Once again, this circuit family is log-space uniform because all of the tables can be generated in log-space by brute force. The circuit has a depth of  $O(\log n)$  because it consists only of table lookups and the addition of  $k$  numbers, both of which can be computed in  $O(\log n)$  depth. Therefore, we have proved theorem 3.4.  $\square$

**Example 3.2 :** Let  $m = 7$  and  $w_1, \dots, w_6 = 1, \dots, 6$ . For  $m = 7$ , we have a generator at  $g = 3$ . Our tables of powers  $T_p$  and our tables of discrete logs  $T_d$  are

$T_p$	
$l$	$g^l$
0	1
1	3
2	2
3	6
4	4
5	5

$T_d$	
$g^l$	$l$
1	0
2	2
3	1
4	4
5	5
6	3

Given these constants, we can compute  $w_1 \cdots w_6 \bmod m$  using the circuit above:

1. We check that  $w_1, \dots, w_6 \neq 0$ .
2. Looking up  $w_i$ 's in  $T_d$ , we get indices  $l = \{0, 2, 1, 4, 5, 3\}$ .

3. Summing up the indices  $l_i$ 's, we obtain

$$L = 0 + 2 + 1 + 4 + 5 + 3 \bmod (7 - 1) = 3.$$

4. We look up  $L$  in  $T_p$  to produce the result  $W = 6$ .

**Example 3.3 :** Let  $m = 8$  and  $w_1, \dots, w_7 = 1, \dots, 7$ . Our tables of powers  $T_p$  and our tables of discrete logs  $T_d$  are

$T_{p1}$	
$l_1$	$(-1)^{l_1}$
0	1
1	7

$T_{p2}$	
$l_2$	$2^{l_2}$
0	1
1	2
2	4
3	0

$T_{p3}$	
$l_3$	$5^{l_3}$
0	1
1	5

$T_d$	
$(-1)^{l_1} 2^{l_2} 5^{l_3}$	$l_1, l_2, l_3$
1	0,0,0
2	0,1,0
3	1,0,1
4	0,2,0
5	0,0,1
6	1,1,1
7	1,0,0

Given these constants, we can compute  $w_1 \cdots w_7 \bmod m$  using the circuit above:

1. We check that  $w_1, \dots, w_7 \neq 0$ .
2. Looking up  $w_i$ 's in  $T_d$ , we get indices

$$l_1 = \{0, 0, 1, 0, 0, 1, 1\}$$

$$l_2 = \{0, 1, 0, 2, 0, 1, 0\}$$

$$l_3 = \{0, 0, 1, 0, 1, 1, 0\}$$

3. Summing up the indices, we obtain  $(L_1, L_2, L_3) = (1, 4, 1)$ .
4. We look up  $(L_1, L_2, L_3)$  in  $T_{p1}, T_{p2}, T_{p3}$  to produce  $(W_1, W_2, W_3) = (-1, 0, 2)$ .
5.  $W = -1 \cdot 0 \cdot 2 \bmod 8 = 0$ .

**Theorem 3.5** *Let  $x, z_1, \dots, z_k$ , where  $k = O(n)$ , be  $n$ -bit CRR integers with a common moduli base  $\mathcal{M} = \langle m_{r-1}, \dots, m_0 \rangle$ . Let  $M$  be the product  $m_{r-1} \cdots m_0$ . We can compute*

1.  $A_4 = (z_1 \cdot z_2 \cdots z_k) \bmod M$
2.  $A_5 = x^{-1} \bmod M$

with  $NC^1$  circuit families.

**Proof :** Let each  $z_i = [z_{(i,r-1)}, \dots, z_{(i,0)}]$ , and let  $A_4 = [A_{(4,r-1)}, \dots, A_{(4,0)}]$ . Then,

$$A_{(4,j)} = \prod_{i=1}^k z_{(i,j)} \bmod m_j. \quad (13)$$

$A_4$  is computed using  $r$  copies of the circuit described in theorem 3.4. This new circuit also has a depth of  $O(\log n)$  and is still log-space uniform. Thus,  $A_4$  can be computed by an  $NC^1$  circuit family.

Let  $x = [x_{r-1}, \dots, x_0]$  and let  $A_5 = [A_{(5,r-1)}, \dots, A_{(5,0)}]$ .  $A_5$  is computed with the same circuit family as  $A_4$  since,

$$A_{(5,j)} = \prod_{i=1}^{m_j-2} x_j \bmod m_j. \quad (14)$$

Therefore, we have proved theorem 3.5. □

## 3.2 Converting from Binary to CRR

**Theorem 3.6** *Let  $x = (x_{n-1}, \dots, x_0)_b$  be a  $n$ -bit binary integer. We can convert  $x$  to  $y = [y_{r-1}, \dots, y_0]$  with the moduli base  $\mathcal{M} = \langle m_{r-1}, \dots, m_0 \rangle$  with an  $NC^1$  circuit family.*

**Proof :** We know from the definition of CRR that

$$y_i = \sum_{j=0}^{n-1} 2^j x_j \text{ mod } m_i. \quad (15)$$

We can compute this equation in  $O(\log n)$  depth given a table of moduli,  $T_m$ , containing  $m_i$ , as follows.

1. Lookup  $m_i$  from our table of moduli  $T_m$ . This can be done in  $NC^1$  via selector trees [1].
2. For all  $i < r, j < n$ , compute  $2^j \text{ mod } m_i$ . We can do this in  $O(\log n)$  depth using the multiplication of  $k$  numbers circuit of the previous section.
3. Compute  $y_i = \sum_{j=0}^{n-1} 2^j x_j \text{ mod } m_i$ . This can be done in  $O(\log n)$  depth with our multiplication and addition of  $k$  numbers circuit from the previous section.

This circuit family is log-space uniform since we can generate a list of primes  $m_i$  in polynomial time and in log-space using the following method.

1. Determine the range that we have to search before we find a sufficient number of primes,  $range = c \cdot n$ .
2. Find the primes: for each integer  $i < range$ , check all  $j < i$ . If  $j$  divides  $i$  then  $i$  is not prime.
3. Finally, output the modulus  $2^j$ , where  $j = O(\log n)$  is chosen such that  $2^j > range$ .

□

**Example 3.6 :** Let  $x = 51 = (0, 0, 1, 1, 0, 0, 1, 1)_b$ .  $T_m = \{2^3, 7, 5\}$ . We can convert  $x$  to CRR using the above circuit:

1. Look up  $m_i$  from our table of moduli:  $m_2 = 8$ ,  $m_1 = 7$ , and  $m_0 = 5$ .
2. Computing  $2^i \text{ mod } m_i$ , we obtain

$m_i$	$2^7$	$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$
8	0	0	0	0	0	4	2	1
7	2	1	4	2	1	4	2	1
5	3	4	2	1	3	4	2	1

3. We compute the CRR representation of  $(0, 0, 1, 1, 0, 0, 1, 1)_b, [y_2, y_1, y_0]$ .

$$\begin{aligned} y_2 &= 0 + 0 + 2 + 1 \bmod 8 = 3 \\ y_1 &= 4 + 2 + 2 + 1 \bmod 7 = 2 \\ y_0 &= 2 + 1 + 2 + 1 \bmod 5 = 1 \end{aligned}$$

## 3.3 Rank and Other Related Problems

### 3.3.1 Computing Rank

Recall the definition of rank from section 2.1.

**Definition 3.1** *Let  $x = [x_{r-1}, \dots, x_0]$  be an  $n$ -bit CRR integer with moduli base  $\mathcal{M}$  and product  $M$ . From the Chinese remainder theorem,*

$$\sum_{i < r} x_i v_i = \mathcal{R}(x, \mathcal{M})M + x \quad (16)$$

The coefficient  $\mathcal{R}(x, \mathcal{M})$  is known as the rank of  $x$  with respect to moduli base  $\mathcal{M}$ .

We now present a method for computing rank from [3]. To simplify discussion, we will use  $\mathcal{R}$  as a shorthand for  $\mathcal{R}(x, \mathcal{M})$ . Let  $c_i = \lfloor (M/m_i)^{-1} \rfloor_{m_i}$ . We first rewrite the Chinese remainder theorem as,

$$\sum_{i < r} c_i x_i / m_i = \mathcal{R} + x/M \quad (17)$$

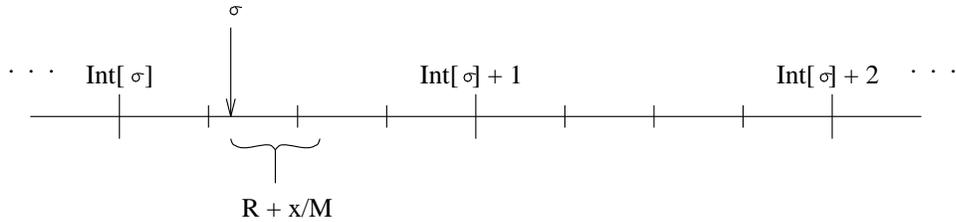
$\mathcal{R}$  has  $O(\log n)$  bits because both  $c_i, x_i < m_i$ . From (17),  $\sum c_i x_i / m_i \leq \sum m_i \leq rm$ , where  $m$  is the largest  $m_i$ . Therefore,  $\mathcal{R}$  has  $O(\log r + \log m) = O(\log n)$  bits.

We calculate  $\mathcal{R}$  by approximating  $\sum c_i x_i / m_i$ . Let  $\sigma_i$  be the integral part plus the first  $q$  bits of the fractional part of  $c_i x_i / m_i$ , where  $r \cdot 2^{-q} < 1/4$ . We approximate  $\sum c_i x_i / m_i$  by  $\sigma = \sum \sigma_i$ . Our approximation error,  $\epsilon$ , is bounded by

$$0 \leq \epsilon = \sum_{i < r} \frac{c_i x_i}{m_i} - \sigma = \sum_{i < r} \left( \frac{c_i x_i}{m_i} - \sigma_i \right) < r \cdot 2^{-q} < \frac{1}{4}.$$

From equation (17),  $\sigma$  is an underapproximation of  $\mathcal{R} + x/M$ . Let  $\text{int}[\sigma]$  denote the integral part of  $\sigma$  and  $\text{frac}[\sigma]$  denote the fractional part of  $\sigma$ . From (17),

$$\mathcal{R} = \sum_{i < r} c_i x_i / m_i - x/M = \text{int}[\sigma] + \text{frac}[\sigma] + \epsilon - x/M \quad (18)$$

Figure 5: When  $\text{frac}[\sigma] < 3/4$ 

From equation (18), we see that  $\text{int}[\sigma]$  is an underapproximation of  $\mathcal{R}$ . Since  $\text{frac}[\sigma] < 1$ ,  $\epsilon < 1/4$ , and  $x/M < 1$ , we have,

$$\begin{aligned} 0 &< \mathcal{R} - \text{int}[\sigma] = \text{frac}[\sigma] + \epsilon - x/M < 5/4 \\ \Rightarrow \quad \mathcal{R} - \text{int}[\sigma] &= 0 \text{ or } 1, \text{ since } \mathcal{R} \text{ and } \text{int}[\sigma] \text{ are integers.} \end{aligned}$$

More precisely,

**Lemma 3.7** *If  $\text{frac}[\sigma] \leq 3/4$ , then  $\mathcal{R} = \text{int}[\sigma]$ .*

**Proof :** If  $\text{frac}[\sigma] \leq 3/4$ , then  $\text{frac}[\sigma] + \epsilon - x/M < 1$ , so  $\mathcal{R} - \text{int}[\sigma] = 0$ . □

**Lemma 3.8** *If  $\text{frac}[\sigma] > 3/4$ , then  $x/M < 1/4$  or  $x/M > 3/4$ .*

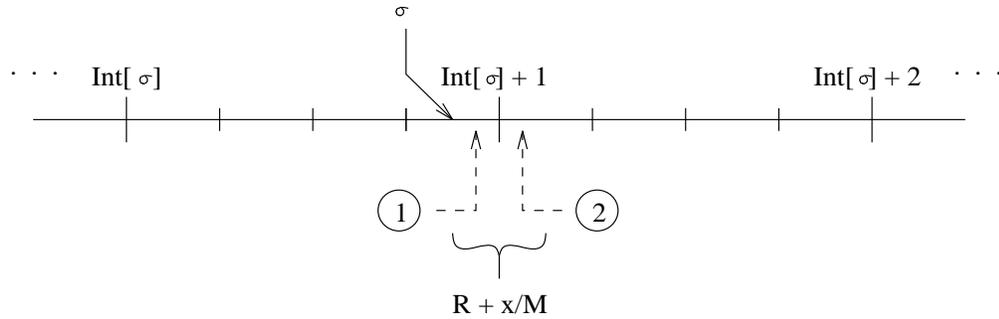
*If  $1/4 \leq x/M \leq 3/4$ , then  $\text{frac}[\sigma] \leq 3/4$ .*

**Proof :** If  $\text{frac}[\sigma] > 3/4$ , then  $3/4 < \text{frac}[\sigma] + \epsilon < 5/4$ , so  $x/M < 1/4$  or  $x/M > 3/4$ .

Likewise, if  $1/4 \leq x/M \leq 3/4$ ,  $1/4 \leq \text{frac}[\sigma] + \epsilon \leq 3/4$ , so  $\text{frac}[\sigma] \leq 3/4$ . □

Lemma 3.7 is illustrated in figure 5. Note that, pictorially, the real value,  $\mathcal{R} + x/M$  is always to the right of  $\sigma$ , because  $\sigma$  is an underapproximation. Also, the rank is always the last integer to the left of  $\mathcal{R} + x/M$  since  $0 \leq x/M < 1$ . We see from figure 5 that if  $\text{frac}[\sigma] < 3/4$ , our approximation  $\sigma$  will never be off by so much that an integer would occur in between  $\sigma$  and  $\mathcal{R} + x/M$ . Instead, both  $\sigma$  and the real value are guaranteed to be within the same unit interval. Therefore, in this case,  $\text{int}[\sigma]$  is the correct value for the rank.

In contrast, in Lemma 3.8, our approximation is close enough to an integer that the integer could occur in between our approximation and  $\mathcal{R} + x/M$ . This is important because if an integer occurs in between, that integer is the true rank instead of our approximated value,  $\text{int}[\sigma]$ . This is illustrated in figure 6. When  $\text{frac}[\sigma] > 3/4$ , we

Figure 6: When  $\text{frac}[\sigma] > 3/4$ 

have two possibilities. The first is when  $\mathcal{R} + x/M$  is within the same unit interval. This occurs when  $x$  is large, ie.  $x/M > 3/4$ . In this case, we know that our approximated rank is correct. The second possibility is that  $\mathcal{R} + x/M$  is within the next unit interval. This occurs when  $x$  is small, ie.  $x/M < 1/4$ . Here the integer which intervenes,  $\text{int}[\sigma] + 1$ , is the correct rank.

Thus, the problem reduces to determining if  $x$  is large or small. If  $x$  is large (possibility 1), we know that our approximated value is in the correct unit interval and  $\text{int}[\sigma]$  is the correct rank. If  $x$  is small (possibility 2), we know that our approximated value is off by one unit interval and hence the correct rank is  $\text{int}[\sigma] + 1$ .

The next lemma provides us with a test to determine if  $x$  is large or small.

**Lemma 3.9** *Let  $\sigma(x)$  be gotten by the approximation method described above for CRR integer  $x$ . Furthermore, let  $\text{frac}[\sigma(x)] > 3/4$ . Then,*

$$\frac{x}{M} > \frac{3}{4} \iff |3^k x|_M \bmod 3 \neq 0,$$

where  $0 < k \leq \log_3 M$  is the smallest integer such that  $\text{frac}[\sigma(|3^k x|_M)] \leq 3/4$ .

**Proof :** We first show that  $k$  exists, by showing an upperbound on  $k$ . If  $\text{frac}[\sigma(x)] > 3/4$ , then by lemma 3.8, either  $x/M < 1/4$  or  $x/M > 3/4$ . If  $x/M < 1/4$ , then let  $k$  be the smallest integer where  $3^k x/M \geq 1/4$ . Clearly,  $k \leq \log_3 M$ . Also,  $3^k x/M \leq 3/4$  since, otherwise,  $k$  would no longer be the *smallest* integer where  $3^k x/M \geq 1/4$ . Therefore,  $1/4 \leq 3^k x/M \leq 3/4$ , so by lemma 3.8,  $\text{frac}[\sigma(|3^k x|_M)] \leq 3/4$ . Similarly,  $k$  exists for  $x/M > 3/4$  (just see that  $|3^k x|_M = M - 3^k y$ , for some  $y/M < 1/4$ ).

Next, we show that the above equivalence holds. Suppose  $x/M < 1/4$  and  $|3^k x|_M \bmod 3 \neq 0$  for the smallest  $k$  where  $\text{frac}[\sigma(3^k x)] \leq 3/4$ . In this case,  $|3^k x|_M =$

$3^k x$  since  $1/4 \leq 3^k x/M \leq 3/4$ . Therefore, we have  $|3^k x|_M \bmod 3 = 3^k x \bmod 3 = 0$ , which is a contradiction. So  $|3^k x|_M \bmod 3 \neq 0 \Rightarrow x/M > 3/4$ .

For the reverse implication, let  $x/M > 3/4$ . Now,  $|3^k x|_M = M - 3^k y$ , for some  $y/M < 1/4$ .  $M - 3^k y \bmod 3 = M \bmod 3 \neq 0$ , since  $M$  contains only factors of the form  $2^j$  or primes greater than 3. Therefore  $x/M > 3/4 \Rightarrow |3^k x|_M \bmod 3 \neq 0$ .  $\square$

This lemma works because we chose our moduli to be primes  $p > 3$  or of the form  $2^j$ . This means that  $M$  is not divisible by 3, and we can detect wraparound by computing mod 3. For example  $|3x|_M \bmod 3$  should be 0 unless  $3x > M$  (or in other words, unless  $x$  is large). Therefore, computing mod 3 allows us to determine if  $x$  is large or small. We multiply by  $3^k$  instead of just 3 to ensure that  $|3^k x|_M$  is within a range where we can compute its rank. This is because we need the rank before we can compute  $|3^k x|_M \bmod 3$ .

Pictorially, we start off in a situation illustrated by figure 6. When we multiply by  $3^k$ , we are, in effect, pulling apart possibilities (1) and (2). Possibility 1 moves left and possibility 2 moves right until they are within the safe areas where we can calculate their ranks and their remainders mod 3.

The above lemmas lead to the following circuit family.

1. Compute  $\sigma(x)$ . This can be done in  $O(\log n)$  depth since all the arithmetic is on  $O(\log n)$  bit integers.
2. If  $\text{frac}[\sigma(x)] \leq 3/4$  then  $\mathcal{R} = \text{int}[\sigma(x)]$ .
3. Otherwise, compute in parallel,  $\sigma(3^k x)$ . We can identify the smallest  $k$  where  $\text{frac}[\sigma(3^k x)] \leq 3/4$  with a tree circuit in  $O(\log n)$  depth.
4. Compute

$$|3^k x|_M \bmod 3 = -(M \cdot \text{int}[\sigma(|3^k x|_M)]) \bmod 3 + \sum y_i \nu_i \bmod 3, \quad (19)$$

where  $[y_{r-1}, \dots, y_0]$  is the CRR of  $3^k x$ . This can be done in  $O(\log n)$  depth using the CRR arithmetic circuits described in theorems 3.1, 3.4 and 3.5. Note that in the above equation,  $M$  and  $v_i$  are never computed, only  $M \bmod 3$  and  $v_i \bmod 3$ . We can compute  $M \bmod 3$  and  $v_i \bmod 3$  since they are both products of small integers, with  $M = \prod m_i$  and  $v_i = M/m_i |(M/m_i)^{-1}|_{m_i}$ .

This equation correctly computes  $\lfloor 3^k x \rfloor_M \bmod 3$  because by definition,  $\text{frac}[\sigma(3^k x)] < 3/4$ . Hence, by lemma 3.8, and 3.7,  $\text{int}[\sigma(3^k x)]$  is the rank of  $3^k x$  and the equation is just a rewrite of the Chinese remainder theorem.

5. If  $\lfloor 3^k x \rfloor_M \bmod 3 = 0$  then  $R = \text{int}[\sigma(x)] + 1$  else  $R = \text{int}[\sigma(x)]$ .

The above circuit is log-space uniform since it is just sequential and parallel combinations of log-space uniform subcircuits. Therefore, we have proved the following theorem.

**Theorem 3.10** *The rank  $R(x, M)$  can be computed by an  $NC^1$  circuit family.*

**Example 3.10 :** Let  $\mathcal{M} = \langle 8, 7, 5 \rangle$  with  $M = 280$ . According to the above discussion, we need to approximate each  $c_i x_i / m_i$  to 4 bits since

$$3 \cdot 2^{-4} < 1/4.$$

We compute the  $c_i$ 's as

$$c_2 = \lfloor (280/8)^{-1} \rfloor_8 = \lfloor 3^{-1} \rfloor_8 = 3$$

$$c_1 = \lfloor (280/7)^{-1} \rfloor_7 = \lfloor 5^{-1} \rfloor_7 = 3$$

$$c_0 = \lfloor (280/5)^{-1} \rfloor_5 = \lfloor 1^{-1} \rfloor_5 = 1$$

Now let  $x = [3, 2, 1] = 51$ . We can compute  $\mathcal{R}(x, \mathcal{M})$  using the above circuit:

1.  $\sigma([3, 2, 1]) = \sigma_2 + \sigma_1 + \sigma_0 = 10.0010_b$ , since

$$\sigma_2 = 3 \cdot 3/8 = 1.0010_b$$

$$\sigma_1 = 3 \cdot 2/7 = 0.1101_b$$

$$\sigma_0 = 1 \cdot 1/5 = 0.0011_b$$

2. Since  $\text{frac}[\sigma] = 0.0010_b \leq 3/4$ ,  $\text{rank } \mathcal{R}(x, \mathcal{M}) = 2$ .

To illustrate lemma 3.9, let  $y = [1, 1, 1] = 1$ . We can compute  $\mathcal{R}(y, \mathcal{M})$  using the above circuit:

1.  $\sigma([1, 1, 1]) = 0.0110_b + 0.0110_b + 0.0011_b = 0.1111_b$ .

2.  $\text{frac}[\sigma] = 0.1111_b > 3/4$ .

3.

$$\begin{aligned} \text{frac}[\sigma([3, 3, 3])] &= \text{frac}[\sigma(3)] = 0.1111_b > 3/4 \\ \text{frac}[\sigma([1, 2, 4])] &= \text{frac}[\sigma(9)] = 0.1111_b > 3/4 \\ \text{frac}[\sigma([3, 6, 2])] &= \text{frac}[\sigma(27)] = 0.0001_b \leq 3/4 \end{aligned}$$

4. We compute  $[3, 6, 2] \bmod 3$  using (19). First, we compute

$$\begin{aligned} \prod m_i \bmod 3 &= 8 \cdot 7 \cdot 5 \bmod 3 = 1 \\ \nu_2 \bmod 3 &= (280/8)|3^{-1}|_8 \bmod 3 = 0 \\ \nu_1 \bmod 3 &= (280/7)|3^{-1}|_7 \bmod 3 = 0 \\ \nu_0 \bmod 3 &= (280/5)|1^{-1}|_5 \bmod 3 = 2 \end{aligned}$$

Applying (19), we obtain

$$[3, 6, 2] \bmod 3 = -(1 \cdot 4) + (3 \cdot 0) + (6 \cdot 0) + (2 \cdot 2) \bmod 3 = 0.$$

5. Hence, the rank  $\mathcal{R}([1, 1, 1], \mathcal{M}) = 0 + 1 = 1$ .

### 3.3.2 The Mod Operation and Moduli Base Extension

In this section, we present a circuit to compute CRR integers modulo small primes, using the rank circuit developed in the last section. Building on this, we then present a circuit which extends the moduli base of CRR integers, allowing us to represent the same value in many different CRR bases. This extension circuit is a key component of our division circuit presented in the next chapter.

**Lemma 3.11** *Let  $x = [x_{r-1}, \dots, x_0]$  be an  $n$ -bit CRR number. Let  $m_r$  be a  $O(\log n)$  bit prime, or of the form  $2^j$ , where  $j = O(\log n)$ . We can compute  $x \bmod m_r$  with an  $NC^1$  circuit family.*

**Proof :**  $x \bmod m_r$  is computed with the following  $NC^1$  circuit family.

1. Calculate the rank  $\mathcal{R}(x)$  using the rank computation circuit.

2.  $x \bmod m_r = -(\mathcal{R} \prod m_i) \bmod m_r + \sum x_i \nu_i \bmod m_r$ . This step involves only  $\bmod m_r$  arithmetic which can be done in  $O(\log n)$  steps using the arithmetic circuits from theorems 3.1, 3.4 and 3.5. Notice that this equation is similar to (19), which we used to compute  $\bmod 3$  in our rank computation circuit.

□

**Example 3.11 :** Let  $\mathcal{M} = \langle 8, 7, 5 \rangle$ ,  $x = [3, 2, 1] = 51$ , and  $m_r = 11 = [3, 4, 1]$ . We can compute  $x \bmod m_r$  using the above circuit:

1.  $\mathcal{R}([3, 2, 1]) = 2$ .

2.

$$\begin{aligned} \prod m_i \bmod m_r &= 8 \cdot 7 \cdot 5 \bmod 11 = 5 \\ \nu_2 \bmod m_r &= (280/8)|3^{-1}|_8 \bmod 11 = 6 \\ \nu_1 \bmod m_r &= (280/7)|5^{-1}|_7 \bmod 11 = 10 \\ \nu_0 \bmod m_r &= (280/5)|1^{-1}|_5 \bmod 11 = 1 \\ \implies x \bmod m_r &= (3)(6) + (2)(10) + (1)(1) - (2)(5) \bmod 11 \\ &= 29 \bmod 11 = 7 \end{aligned}$$

**Definition 3.2** *If  $\mathcal{M} = \langle m_{r-1}, \dots, m_0 \rangle$  is a CRR moduli base, and  $\{m_s, \dots, m_r\} \cap \{m_{r-1}, \dots, m_0\} = \emptyset$ , then  $\mathcal{M}\langle m_s, \dots, m_r \rangle = \langle m_s, \dots, m_r, m_{r-1}, \dots, m_0 \rangle$ . If  $[x_{r-1}, \dots, x_0]$  is the CRR of  $x < M$  in  $\mathcal{M}$ , then the moduli base extension problem is to compute the CRR of  $x = [x_s, \dots, x_r, x_{r-1}, \dots, x_0]$  in  $\mathcal{M}\langle m_s, \dots, m_r \rangle$ .*

We are interested in the base extension problem for two reasons. First, CRR arithmetic corresponds to our normal notion of arithmetic only when the result is less than  $M$ , the product of the moduli. Therefore, we have to extend our moduli base if we want to represent larger numbers or represent with greater precision. Second, some properties in CRR allow us to efficiently compute results, but in a reduced moduli base. In order to use these properties in our circuit designs, we need an efficient method to extend the moduli base back to the full base used in our inputs.

**Theorem 3.12** *Moduli base extension is in  $NC^1$ .*

**Proof :** We compute each  $x_i = x \bmod m_i$ , in parallel, for  $i \in \{m_s, \dots, m_r\}$  using the  $NC^1$  circuit family described in Lemma 3.11.  $\square$

**Example 3.12 :** Let  $\mathcal{M} = \langle 8, 7, 5 \rangle$  and  $\mathcal{M}' = \langle 11, 8, 7, 5 \rangle$ . Furthermore, let  $x$  in  $\mathcal{M}$  be represented as  $[3, 2, 1] = 51$ . Now  $x$  in  $\mathcal{M}'$  is simply  $[7, 3, 2, 1]$  since  $x \bmod 11 = 7$ .

### 3.3.3 Comparison

**Theorem 3.13** *Given  $x$  and  $y$ ,  $n$ -bit CRR numbers we can compute  $x > y$  with an  $NC^1$  circuit family.*

**Proof :** Let  $H = \lceil M/2 \rceil$ . We know that a number  $x$  is in the top half of the number range if  $x + H$  overflows. Since  $M \bmod 3 \neq 0$ , we can detect this overflow by comparing  $|x + H|_3$  with  $|x|_3 + |H|_3$ . If  $x + H > M$ , then  $|x + H|_3$  is actually  $|x + H - M|_3 = |x|_3 + |H|_3 - |M|_3 \neq |x|_3 + |H|_3$ . Therefore, if  $|x + H|_3 \neq |x|_3 + |H|_3$ , then  $x$  is in the top half of the number range.

These ideas lead to the following circuit.

1. Lookup  $H$ , the smallest number such that  $2H > M$ .
2. Compute if X is greater than H: If  $|x + H|_3 \neq (|x|_3 + |H|_3) \bmod 3$  then  $x > H = \text{true}$ .
3. Compute if Y is greater than H: If  $|y + H|_3 \neq (|y|_3 + |H|_3) \bmod 3$  then  $y > H = \text{true}$ .
4. If  $x > H$  and not  $(y > H)$  then return  $(x > y)$
5. If  $y > H$  and not  $(x > H)$  then return  $(x < y)$
6. If  $|x - y|_3 = (|x|_3 - |y|_3) \bmod 3$  then return  $(x > y)$  else return  $(x < y)$ .

Steps (2), (3), and (6) have  $O(\log n)$  depth using the mod circuit described in lemma 3.11. The remaining steps are just small arithmetic and logic circuits, and also have  $O(\log n)$  depth. All of the steps involve log-space uniform circuits. Hence, we have shown that comparison is in  $NC^1$ .  $\square$

**Example 3.13 :** Let  $\mathcal{M} = \langle 8, 7, 5 \rangle$ ,  $x = [3, 2, 1] = 51$ , and  $y = [3, 4, 1] = 11$ . We can compare  $x$  and  $y$  with the circuit above:

1. Look up  $H = [4, 0, 0]$ .
2.  $X$  is less than  $H$  since  $|x + H|_3 = 2 = |x|_3 + |H|_3$ .
3.  $Y$  is less than  $H$  since  $|y + H|_3 = 1 = |y|_3 + |H|_3$ .
- 4.
- 5.
6.  $|x - y|_3 = 1 = |x|_3 - |y|_3$ . Therefore  $x > y$ .

## Chapter 4

### Division

In this chapter, we show that division is in  $NC^1$ . In section 4.1, we discuss two subproblems of CRR division — scaling and shifting. Scaling is similar to the general division problem except that the divisor is restricted to products of some of the moduli in our CRR moduli base. In CRR shifting, we are interested in dividing by a  $2^k$  which is smaller than the largest moduli.

In section 4.2, we summarize the division circuit presented in [1]. Although their circuit has  $O(\log n)$  depth, it is not log-space uniform, only P-uniform (which is a more relaxed definition of uniformity). Finally, in section 4.3, we modify the circuit of section 4.2 to produce a log-space uniform,  $O(\log n)$  depth circuit for division. Our circuit reduces the problem of general division to CRR scaling and shifting which we then solve with the circuit in section 4.1

### 4.1 Two Sub-problems of Division

#### 4.1.1 CRR Scaling

**Definition 4.1** *Let  $x$  be a  $n$ -bit CRR integer with moduli base  $\mathcal{M}$ . Let  $\mathcal{D} = \langle d_{s-1}, \dots, d_0 \rangle$  be a proper, nonempty subset of  $\mathcal{M}$ . The CRR scaling problem is to compute*

$$z = \lfloor \frac{x}{D} \rfloor, \tag{20}$$

where  $D = \prod_{i < s} d_i$ .

The CRR scaling problem is an important restricted case of general division. It is analogous to division by a power of 2 in binary. When we describe our circuit for

division later in section 4.3, we will reduce the problem to division by some product of the moduli. Hence we need a circuit to perform CRR scaling in  $O(\log n)$  depth. The circuit we present here is a parallel adaptation of some ideas from [13].

Scaling can be computed in  $NC^1$  by using base extension. Consider the following example. Let  $\mathcal{M} = \langle 8, 7, 5 \rangle$ ,  $x = [3, 2, 1] = 51$ , and  $\mathcal{D} = \langle 5 \rangle$ . In this case  $D = 5$ , and our problem is to evaluate  $\lfloor 51/5 \rfloor$ . Let  $\mathcal{M}' = \mathcal{M} - \mathcal{D} = \langle 8, 7 \rangle$ .

1. Compute the remainder  $R = x \bmod D$  in  $\text{CRR}(\mathcal{D})$ .  $R$  can be adequately represented in  $\mathcal{D}$  because  $R < D$ .

$$R = x \bmod D = 1 = [1]_{\mathcal{D}}$$

2. Extend the remainder  $R$  back to  $\text{CRR}(\mathcal{M})$ :  $R = [1, 1, 1]_{\mathcal{M}}$ .
3. Subtract  $R$  from  $x$  to make  $x$  a multiple of  $D$

$$x - R = [3, 2, 1] - [1, 1, 1] = [2, 1, 0]$$

4. Find  $D^{-1}$  in  $\text{CRR}(\mathcal{M}')$ .  $D^{-1}$  is the modular inverse and not the arithmetic inverse.

$$D^{-1} \bmod (M/D) = [5, 5]^{-1} \bmod (M/D) = [5, 3]_{\mathcal{M}'}$$

5. Multiply  $(x - R)$  by  $D^{-1}$  to obtain  $\lfloor x/D \rfloor$  in  $\text{CRR} \mathcal{M}'$ . Because  $(x - R)$  is a perfect multiple of  $D$ ,  $(x - R)D^{-1} = (x - R)/D$ .

$$(x - R)D^{-1} \bmod (M/D) = [2, 1][5, 3] = [2, 3]$$

6. Extend the result back to  $\text{CRR}(\mathcal{M})$ :  $\lfloor x/y \rfloor = [2, 3, 0]$ .

**Theorem 4.1** *CRR scaling can be computed in  $NC^1$ .*

**Proof :** The proof follows the steps in the example. Let  $\mathcal{M}' = \mathcal{M} - \mathcal{D}$ . We make the observation that if  $\mathcal{N} \subseteq \mathcal{M}$ , then the  $\text{CRR}(\mathcal{N})$  vector for  $y$  is obtained from the  $\text{CRR}(\mathcal{M})$  vector for  $y$  by simply deleting the entries corresponding to moduli in  $\mathcal{M} - \mathcal{N}$ . This deletion is clearly in  $NC^1$ .

1. Compute the remainder  $R = x \bmod D$  as  $R = [|x|_{d_{s-1}}, \dots, |x|_{d_0}]$  in the  $\text{CRR} \mathcal{D}$ .

2. Extend  $R$  from  $\mathcal{D}$  back to  $\mathcal{M}$ .
3. Compute  $x - R$  in  $\mathcal{M}$ .
4. Compute  $D^{-1} \bmod (M/D)$  in  $\mathcal{M}'$ .  $D^{-1}$  exists because  $D$  and  $M/D$  are relatively prime.
5. Compute  $z = (x - R)D^{-1} \bmod (M/D)$  in  $\mathcal{M}'$ . Since  $(x - R)$  is divisible by  $D$ ,  $(x - R)D^{-1}$  is equivalent to  $\lfloor x/D \rfloor$ .
6. Extend  $z$  to  $\mathcal{M}$ .

Step (1) is  $NC^1$  by our observation. Steps (2) and (6) are  $NC^1$  by theorem 3.12, and the remaining steps are  $NC^1$  by theorems 3.1 and 3.5. Therefore, we have proved that CRR scaling is in  $NC^1$ .  $\square$

### 4.1.2 CRR Shifting

**Definition 4.2** *Let  $x$  be an  $n$ -bit CRR integer with moduli base  $\mathcal{M} = \langle m_{r-1}, \dots, m_0 \rangle$ . Let  $m_r$  be our modulus of the form  $2^j$ , where  $j = O(\log n)$ . The CRR shifting problem is to compute  $z = \lfloor x/2^k \rfloor$ , where  $0 < k < j$ .*

**Theorem 4.2** *CRR shifting is in  $NC^1$ .*

**Proof :**  $z = \lfloor x/2^k \rfloor$  is computed with the following  $NC^1$  circuit family. Let  $m_{r-1} \in \mathcal{M}$  be the modulus of the form  $2^j$ , and  $x_{r-1}$  be the corresponding digit, ie.  $x_{r-1} = \lfloor x \rfloor_{m_{r-1}}$ . Furthermore, let  $\mathcal{M}' = \mathcal{M} - \{m_{r-1}\} + \{2^{j-k}\}$ .

1. Compute the remainder  $R = x \bmod 2^k$ . This can be computed in  $O(\log n)$  depth because  $R = x_{r-1} \bmod 2^k$ , with both  $x_{r-1}$  and  $2^k$  being  $O(\log n)$  bit integers.
2. Compute  $z = [z_{r-1}, \dots, z_0]$  in  $\text{CRR}(\mathcal{M}')$ .

$$z_i = (x_i - R)(2^k)^{-1} \bmod m_i, \text{ for } i = 0, \dots, r-2. \quad (21)$$

$$z_{r-1} = \lfloor x_{r-1}/2^k \rfloor \bmod 2^{j-k} \quad (22)$$

$z$  is computed in moduli base  $\mathcal{M}'$  and not  $\mathcal{M}$  because  $m_i$  and  $2^k$  are relatively prime only for  $i = 0, \dots, r-2$ . So, we can divide using the modular inverse

only for  $m_0, \dots, m_{r-2}$ . For  $m_{r-1}$ , we have to use (22) instead. Equation (22) works because

$$\begin{aligned} z \bmod 2^j &= x_{r-1} \\ z &= 2^j a + x_{r-1} \\ \lfloor z/2^k \rfloor &= 2^{j-k} a + \lfloor x_{r-1}/2^k \rfloor \\ &= \lfloor x_{r-1}/2^k \rfloor \pmod{2^{j-k}} \end{aligned}$$

This step can be performed in  $O(\log n)$  depth because equations (21) and (22) involve only  $O(\log n)$  bit numbers.

3. Extend  $z$  to  $M$ . This is  $NC^1$  by theorem 3.12.

Therefore, we have proved theorem 4.2. □

**Example 4.2 :** Let  $\mathcal{M} = \langle 8, 7, 5 \rangle$ ,  $x = [3, 2, 1] = 51$  and  $2^k = 4$ . We can compute  $\lfloor [3, 2, 1]/4 \rfloor$  using the circuit described above.

1.  $R = [3, 2, 1] \bmod 4 = 3 \bmod 4 = 3$ .
2. We compute  $z = [z_2, z_1, z_0]$  in  $\langle 2, 7, 5 \rangle$ .

$$\begin{aligned} z_0 &= (1 - 3) \cdot 4^{-1} \bmod 5 = 2 \\ z_1 &= (2 - 3) \cdot 4^{-1} \bmod 7 = 5 \\ z_2 &= \lfloor 3/4 \rfloor = 0 \end{aligned}$$

3. We extend  $[0, 5, 2]$  in  $\langle 2, 7, 5 \rangle$  back to the full base of  $\langle 8, 7, 5 \rangle$ .

$$z_{\mathcal{M}} = [4, 5, 2] = 12.$$

## 4.2 A P-Uniform Division Circuit

The first  $O(\log n)$  depth circuit for general division was presented by Beame, Cook and Hoover in [1]. Their circuit was not log-space uniform, only  $P$ -uniform (ie the

circuit can be generated in polynomial time). However, since our  $NC^1$  circuit family for division (which will be presented in section 4.3) is based on their design, we describe their circuit in detail here.

The main idea of most  $O(\log n)$  depth division circuits is to compute an approximation to the reciprocal of the divisor using the well-known geometric series,

$$1/y = 1 + (1 - y) + \cdots + (1 - y)^i + \cdots, \text{ when } 0 < y < 1 \quad (23)$$

**Definition 4.3**  $\bar{a}$  is an  $n$ -bit under-approximation to  $a$  if

$$0 \leq a - \bar{a} \leq 2^{-n}$$

We compute  $\overline{1/y}$ , an  $n$ -bit under-approximation to  $1/y$  by

$$\overline{1/y} = 1 + (1 - y) + (1 - y)^2 + \cdots + (1 - y)^n, \text{ for } 1/2 \leq y < 1 \quad (24)$$

To see that this is indeed an under-approximation, consider

$$\begin{aligned} 1/y - \overline{1/y} &= \sum_{i \geq 0} (1 - y)^i - \sum_{0 \leq i \leq n} (1 - y)^i \\ &= \sum_{i > n} (1 - y)^i \\ &\leq \sum_{i > n} 2^{-i} \\ &= 2^{-n} \end{aligned}$$

**Lemma 4.3** Let  $\overline{D/y}$  be an  $n$ -bit under-approximation to the reciprocal of  $y/D$  for some positive integer  $D$ . Then, for  $x \leq 2^n$ ,

$$\lfloor \frac{x}{y} \rfloor - \lfloor \frac{x \cdot \overline{D/y}}{D} \rfloor \leq 1$$

**Proof :**

$$\begin{aligned} \lfloor \frac{x \cdot \overline{D/y}}{D} \rfloor &\geq \lfloor \frac{x \cdot (D/y - 2^{-n})}{D} \rfloor \\ &= \lfloor x/y - x2^{-n}/D \rfloor \\ &\geq \lfloor x/y \rfloor - 1 \end{aligned}$$

□

We now present the circuit from [1] to compute  $\lfloor x/y \rfloor$ . The circuit first scales  $y$  to within the range specified by equation (24) so that we may apply the approximation. The above lemma tells us that if we have an  $n$ -bit under-approximation to the reciprocal of a scaled  $y$ ,  $(\overline{D/y})$ , our result when multiplied by  $x$  and rescaled by  $D$  will be within 1 of  $\lfloor x/y \rfloor$ . The circuit uses arithmetic in the  $n^2$  bit CRR( $\mathcal{D}$ ) to calculate powers of  $t = (1 - y)$ .

1. Determine  $d \geq 2$  such that  $2^{d-1} \leq y < 2^d$  and compute  $t = 1 - y2^{-d}$ .
2. Evaluate  $t^i$ ,  $i = 0, \dots, n$  by converting the numerator  $t$  to CRR( $\mathcal{D}$ ) and multiplying using the circuits described in theorems 3.1 and 3.5.
3. Convert results back to binary using a Chinese remaindering circuit.
4. Compute  $\overline{2^d/y} = 1 + t + \dots + t^n$ .
5. Compute  $z = 2^{-d}x \cdot \overline{2^d/y}$ .
6. Calculate  $x - yz$  and determine whether  $\lfloor x/y \rfloor$  is  $z$  or  $z + 1$ .

All of these steps except step (3) can easily be computed with log-space uniform circuits of  $O(\log n)$  depth. Step (3) is more difficult. In [1], they gave a Chinese remaindering circuit for step (3) that is not log-space uniform, only P-uniform (ie. their circuit can be generated in polynomial time). Their circuit for Chinese remaindering is as follows:

1. Compute basis vectors  $\nu_i$  in binary.
2. Compute the binary value of  $x$ ,  $\text{CRT}(x)$ , using the Chinese remainder theorem.
3. Estimate the rank  $\mathcal{R}(x, \mathcal{M})$  using large numbers.
4. Determine  $\text{Binary}(x) = \text{CRT}(x) - \mathcal{R}(x, \mathcal{M})M$ .

The main reason this circuit is P-uniform and not log-space uniform is because of the difficulty in computing the  $O(n)$  bit constants for the basis vectors  $\nu_i$  and  $M$ . We will avoid this problem in the next section by first working with an all CRR division circuit, where both the circuit inputs and outputs are in CRR.

**Example 4.2 :** Let  $x = 51$  and  $y = 11$  be 6-bit integers. Let

$$\mathcal{D} = \langle 32, 31, 29, 23, 19, 17, 13, 11, 7, 5 \rangle$$

be our  $n^2$  bit division moduli base. We can calculate  $\lfloor x/y \rfloor$  using the circuit above.

1.  $d = 4$  since  $2^3 \leq 11 < 2^4$ .  $t = 5/16$ .
2. Converting the numerator of  $t$  to CRR, we compute

$$\begin{aligned} t^1 &= [5, 5, 5, 5, 5, 5, 5, 5, 5, 0] \\ t^2 &= [25, 25, 25, 2, 6, 8, 12, 3, 4, 0] \\ t^3 &= [29, 1, 9, 10, 11, 6, 8, 4, 6, 0] \\ t^4 &= [17, 5, 16, 4, 17, 13, 1, 9, 2, 0] \\ t^5 &= [21, 25, 22, 20, 9, 14, 5, 1, 3, 0] \\ t^6 &= [9, 1, 23, 8, 7, 2, 12, 5, 1, 0] \end{aligned}$$

3. Converting back to binary, we obtain

$$\begin{aligned} t^1 &= 0.3125 \\ t^2 &= 0.097656 \dots \\ t^3 &= 0.030517 \dots \\ t^4 &= 0.009536 \dots \\ t^5 &= 0.002980 \dots \\ t^6 &= 0.000931 \dots \end{aligned}$$

$$4. \overline{16/11} = t^1 + t^2 + \dots + t^6 = 1.45412$$

$$5. 51/11 = 1/16 \cdot (51 \cdot \overline{16/11}) = 4.635.$$

$$6. x - yz = 51 - 4(11) = 6 < 11 \text{ so } \lfloor x/y \rfloor = 4.$$

### 4.3 Log-Space Uniform Division

In this section, we present an  $NC^1$  circuit family for division. Please note that all the circuits shown here are not optimized with respect to size and depth. Instead, they are described in a way so they are more easily seen to be log-space uniform and have  $O(\log n)$  depth. Because of the large number of variables used in explaining our circuit, we list below the types of variables used and their functions.

- $x$  and  $y$  are the numerator and denominator in our input.
- $t$ 's ( $t, t_i, T_i$ ) are the intermediates in our approximation process.
- $D$ 's ( $D_i, D$ ) are the scaling and shifting factors applied before our approximation equation is applicable.
- $\mathcal{D} = \langle d_{s-1}, \dots, d_0 \rangle$  is the extended moduli base that allows us to compute without wraparound.

We start off with a circuit for CRR division, which is similar to the integer division problem except that we accept inputs  $x$  and  $y$  in CRR and return the result of  $\lfloor x/y \rfloor$  in CRR. The CRR division circuit is similar to the one used by [1] in the last section, except that the geometric series,

$$1/y \approx 1 + t + \dots + t^n, \text{ where } 1/2 \leq y < 1 \text{ and } t \approx 1 - y \quad (25)$$

yielding an  $n$ -bit under-approximation to the reciprocal of the divisor  $y$  is replaced by a series based on scaling and shifting. First, we partition  $\mathcal{D}$  into  $n$  non-overlapping subsets, with corresponding relatively prime products  $D_1, \dots, D_n$ . Using these  $D_i$ 's as denominators, we then represent the value  $t$  with  $n$  different fractions,  $t_1/D_1, \dots, t_n/D_n$ . Using the product of moduli as our denominators is as natural for CRR as  $2^d$  was for binary since we have corresponding circuits from section 4.1 to compute scaling and shifting in CRR.

To calculate the term  $t_i$ , we multiply together  $i$  of these fractions  $\frac{t_1 \dots t_i}{D_1 \dots D_i}$  so that our result still has some product of moduli as its denominator. This is because we know how to divide by some product of moduli (using CRR scaling and shifting), but not how to divide by some product of *powers* of moduli. We also need to change

the CRR moduli base to an  $O(n^2)$  bit moduli base  $\mathcal{D}$ , so that we can represent the numerator and denominator of  $t_i$  without wraparound.

Next, we present a log-space uniform,  $O(\log n)$  depth Chinese remaindering circuit to convert the results from CRR division back to binary. Finally, we use these two sub-circuits to show that binary division is in  $NC^1$ .

We first start off with a lemma detailing how close our  $t_i/D_i$ 's have to be to  $1 - y$  in order for our approximation to be correct.

**Lemma 4.4** *Let  $1/2 \leq y < 1$  and let  $t_1/D_1, \dots, t_{n+1}/D_{n+1}$  be  $2n$ -bit under-approximations to  $t = 1 - y$ . Then,*

$$\overline{1/y} = 1 + \frac{t_1}{D_1} + \frac{t_1 t_2}{D_1 D_2} + \dots + \prod_{i=1}^{n+1} \frac{t_i}{D_i} \quad (26)$$

is an  $n$ -bit under-approximation to  $1/y$ .

**Proof :** Let  $T = t - 2^{-2n} \leq t_i/D_i$ . Now  $T^j \leq \prod_{i=1}^j t_i/D_i$ . We have

$$1/y = 1 + t + t^2 + \dots \geq \overline{1/y} \geq 1 + T + T^2 + \dots + T^{n+1}$$

Looking at each term,

$$\begin{aligned} t^i - T^i &\leq t^i - (t - 2^{-2n})^i \\ &= t^i - (t^i - i t^{i-1} 2^{-2n} + i(i-1) t^{i-2} 2^{-4n} - \dots) \\ &\leq i t^{i-1} 2^{-2n}, \text{ by the Leibnitz alternating series convergence criterion} \\ &\leq i 2^{-2n}, \text{ since } t \leq 1/2 \end{aligned}$$

Therefore, for sufficiently large  $n$ ,

$$\begin{aligned} 1/y - \overline{1/y} &\leq 2^{-2n} \sum_{i=1}^{n+1} i + \sum_{i>n+1} t^i \\ &\leq 2^{-2n} (n+1)^2 + 2^{-(n+1)} \\ &\leq 2^{-n} \end{aligned}$$

Hence,  $\overline{1/y}$  is an  $n$ -bit under-approximation to  $1/y$ . □

Given  $x$  and  $y$  be  $n$ -bit CRR integers with moduli base  $\mathcal{M} = \langle m_{r-1}, \dots, m_0 \rangle$ , the CRR division problem is to compute  $\lfloor x/y \rfloor$  in CRR.

**Theorem 4.5** *CRR division is in NC<sup>1</sup>.*

**Proof :** We compute  $\overline{1/y}$  in CRR using equation (26) in Lemma 4.4. As mentioned above, we need a  $O(n^2)$  bit CRR system,  $\text{CRR}(\mathcal{D} = \langle d_{s-1}, \dots, d_0 \rangle)$  to accurately represent the numerator and denominator of the largest term. Let  $D_i = \prod_{k=2^{(i-1)n+1}}^{2in} d_k$  and  $D_{i \rightarrow j} = D_i \cdots D_j$ . Notice that each  $D_i > 2^{2n}$  and so can serve as denominators for terms  $t_1/D_1, \dots, t_{n+1}/D_{n+1}$  (our  $2n$ -bit under- approximations for  $t$ ). Our algorithm is as follows.

1. Find  $D = 2^j \prod_{k=1}^{j'} m_k$  such that

$$0 < t = 1 - y/D \leq 1/2$$

We do this by first comparing  $y$  in parallel against  $\prod_{k=0}^{j'} m_k$  and selecting the last  $j'$  such that  $y > \prod_{k=0}^{j'} m_k$ . Next, we compare  $y$  in parallel against  $2^j \prod_{k=0}^{j'} m_k$  and select the first  $j$  such that  $y < 2^j \prod_{k=0}^{j'} m_k$ . The total depth of this step is  $O(\log n)$  with the CRR comparison circuit described in theorem 3.13.

2. Find  $[t_1]_{\mathcal{D}}, \dots, [t_{n+1}]_{\mathcal{D}}$  such that

$$\frac{t_1}{D_1}, \dots, \frac{t_{n+1}}{D_{n+1}}$$

are  $2n$ -bit under-approximations to  $t$ . This step is done in  $O(\log n)$  depth by extending the moduli base of  $y$  to  $\mathcal{D} \cup \mathcal{M}$  and then computing,

$$t_i = \lfloor (D - y)D_i/D \rfloor \tag{27}$$

using the CRR scaling and shifting circuits described in section 4.1.

3. Compute the numerator  $N$  of  $\overline{D/y}$  using the  $n + 1$  term summation,

$$N = (\overline{D/y})D_{1 \rightarrow n+1} = D_{1 \rightarrow n+1} + D_{2 \rightarrow n+1}t_1 + \cdots + \prod_{i=1}^{n+1} t_i \tag{28}$$

This computation uses just ordinary CRR arithmetic in the  $n^2$  bit  $\text{CRR}(\mathcal{D})$  and so requires  $O(\log n)$  depth.

4. Compute  $\overline{x/y} = \frac{x \cdot \overline{D/y}}{D}$  for an approximation to  $\lfloor x/y \rfloor$ . We do this by computing,

$$\overline{x/y} = \frac{Nx}{DD_{1 \rightarrow n+1}} \quad (29)$$

using CRR arithmetic, base extension, scaling and shifting circuits.

5. Finally, if  $x - \overline{x/y} \cdot y < y$  then  $\lfloor x/y \rfloor = \overline{x/y}$  else  $\lfloor x/y \rfloor = \overline{x/y} + 1$

□

**Example 4.5 :** Let  $\mathcal{M} = \langle 8, 7, 5 \rangle$  be an 6-bit CRR. Let  $x = [3, 2, 1] = 51$  and  $y = [3, 4, 1] = 11$ . Let

$$\mathcal{D} = \langle 128, 79, 73, 71, 67, 61, 59, 53, 47, 43, 41, 37, 31, 29, 23, 19, 17, 13, 11, 7, 5 \rangle$$

be our  $n^2$  bit division moduli base. We now have to choose our denominators  $D^i$ . We will choose them in a slightly different way from theorem 4.5, so that we have more manageable numbers to work with. In any case, the algorithm works independently of how the  $D_i$ 's are chosen, as long as  $D_i > 2^{2n}$ . The theorem chooses the  $D_i$  in the simplest way by just sequentially partitioning  $\mathcal{D}$ . However, that method results in grossly large  $D_i$ 's which may make the example difficult to follow. Therefore, to clearly demonstrate how the actual algorithm works, we will choose the  $D_i$ 's so that each  $D_i$  is just large enough to satisfy  $D_i > 2^{2n}$ .

$$\begin{aligned} D_1 &= \langle 73, 71 \rangle = 6557 > 2^{12} \\ D_2 &= \langle 67, 61, 5 \rangle = 20435 > 2^{12} \\ D_3 &= \langle 59, 53, 7 \rangle = 21889 > 2^{12} \\ D_4 &= \langle 47, 43, 11 \rangle = 22231 > 2^{12} \\ D_5 &= \langle 41, 37, 13 \rangle = 19721 > 2^{12} \\ D_6 &= \langle 31, 29, 17 \rangle = 15283 > 2^{12} \\ D_7 &= \langle 79, 23, 19 \rangle = 34523 > 2^{12} \end{aligned}$$

We can calculate  $\lfloor x/y \rfloor$  in CRR using the circuit above:

1.  $D = 2^2 \cdot 5$  since  $0 < 1 - 11/20 \leq 1/2$ .

2. We compute each  $t^i$  via (27).

$$\begin{aligned}
 t_1 &= 9 \cdot 6557/20 = 2332 \\
 t_2 &= 9 \cdot 20435/20 = 9195 \\
 t_3 &= 9 \cdot 21889/20 = 9850 \\
 t_4 &= 9 \cdot 22231/20 = 10003 \\
 t_5 &= 9 \cdot 19721/20 = 8874 \\
 t_6 &= 9 \cdot 15283/20 = 6877 \\
 t_7 &= 9 \cdot 34523/20 = 15535
 \end{aligned}$$

3.

$$\begin{aligned}
 N &= D_{1 \rightarrow n+1} + D_{2 \rightarrow n+1} t_1 + \cdots + \prod_{i=1}^{n+1} t_i \\
 &= 973317470283354092626290031515.
 \end{aligned}$$

4.  $\overline{51/11} = N/(20 \cdot D_{1 \rightarrow n+1}) = 4.628$

5.  $x - \lfloor x/y \rfloor y = 6 < 11$ , so  $\lfloor x/y \rfloor = [4, 4, 4]$ .

We now present a Chinese remaindering circuit for converting the results from the CRR division circuit back to binary. To re-iterate the definition of Chinese remaindering: Given a CRR integer  $x$ , return the binary value of  $x$ .

**Theorem 4.6** *Chinese remaindering is in  $NC^1$*

**Proof :** Let  $y = (y_{n-1}, \dots, y_0)$  be the binary representation of the CRR integer  $x$ . Using the CRR division circuit above, we compute, in parallel,

$$y_i = \lfloor x/2^i \rfloor - 2\lfloor x/2^{i+1} \rfloor. \quad (30)$$

The CRR division circuit always returns either the result of  $[0, 0, \dots, 0]$  or  $[1, 1, \dots, 1]$  which we can then easily convert to binary as 0 or 1 respectively. The depth of this circuit is precisely the depth of CRR division which is  $O(\log n)$ . The circuit can also trivially be seen to be log-space uniform.  $\square$

**Example 4.6 :** Let  $\mathcal{M} = \langle 8, 7, 5 \rangle$ . Let  $x = [4, 4, 4] = 4$  and  $y = (y_2, y_1, y_0)$  be the binary representation of  $x$ . Using the above circuit, we compute

$$y_0 = [4, 4, 4] - 2[2, 2, 2] = [0, 0, 0] = 0$$

$$y_1 = [2, 2, 2] - 2[1, 1, 1] = [0, 0, 0] = 0$$

$$y_2 = [1, 1, 1] - 2[0, 0, 0] = [1, 1, 1] = 1$$

Therefore,  $y = (1, 0, 0)_b$ .

We now state our main result

**Theorem 4.7** *Division, powering and iterated product are all in  $NC^1$*

**Proof :** We present an  $NC^1$  circuit for division below. By theorem 2.4, powering and iterated product are also in  $NC^1$ .

1. Convert binary integers  $x = (x_{n-1}, \dots, x_0)$  and  $y = (y_{n-1}, \dots, y_0)$  to CRR numbers. This can be done in  $O(\log n)$  steps using the circuit from theorem 3.6.
2. Use the CRR division circuit to compute  $\lfloor [x/y] \rfloor_{\mathcal{M}}$ .
3. Use the Chinese remaindering circuit to compute  $\lfloor x/y \rfloor$ .

□

**Corollary 1** *Division, powering and iterated product are all in log-space.*

**Proof :** This follows at once from Theorem 4.7 and Theorem 2.3.

□

## Chapter 5

### $U_E$ Uniformity

In this chapter, we show that division is in  $NC^1$  under  $U_E$  uniformity. Recall from section 2.3,  $U_E$  uniformity is a newer definition for uniformity from [11]. It is more restrictive than the definition we have been using (known as  $U_B$  uniformity). In section 5.1, we formally describe the  $U_E$  uniformity model. Then, in section 5.2, we show that the division circuit family, described in section 4.3, is  $U_E$  uniform. In particular, we show that the prime tables and discrete log tables are computable in logarithmic time and space.

### 5.1 Uniform Circuits and Alternating Turing Machines

**Definition 5.1** *An alternating Turing machine (ATM) is a Turing machine with the addition of “existential” and “universal” states. An existential state is an accepting configuration if one of its successors is accepting. A universal state is an accepting configuration if all of its successors are accepting.*

ATMs are generalizations of non-deterministic Turing machines (NTM). Note that the computations for an ATM with only existential and no universal states is exactly the same as that for an NTM.

$U_E$  uniformity is concerned with the local connections from any gate instead of generating the global structure. We first formalize the notion of local connections [11]. Let  $C = \{C_i\}$  be a family of circuits,  $g$  be a gate in a circuit  $C_n$ , and  $p = \{\text{self}\} \cup \{L, R\}^*$  denote a path. Let  $g(p)$  represent the gate reached by following the

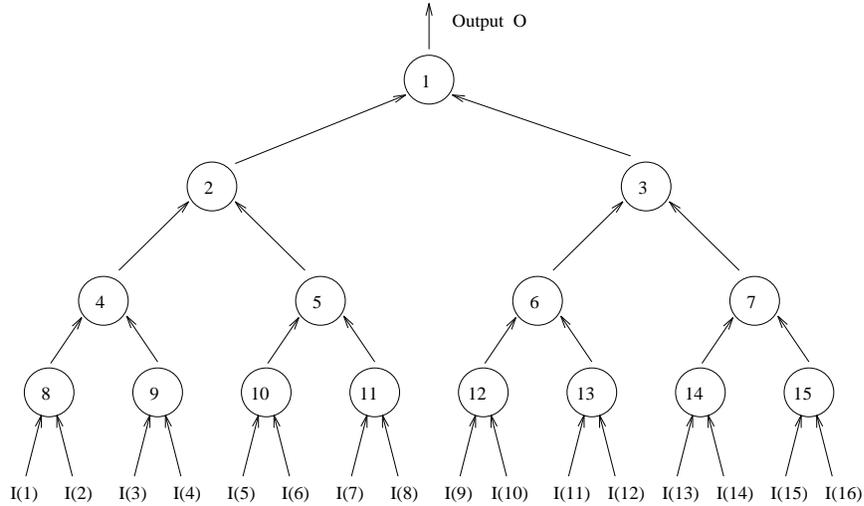


Figure 7: A tree AND circuit for  $n = 16$ .

path  $p$  of inputs to  $g$ . For example  $g(\text{self}) = g$ ,  $g(R)$  is  $g$ 's right input and  $g(RL)$  is  $g$ 's right input's left input. In figure 7,  $0(\text{self})=0$ ,  $0(R) = 3$ , and  $0(RL) = 6$ .

**Definition 5.2** *The extended connection language of a circuit family  $C$ ,  $L_{EC}$  is the set of strings of the form  $\langle n, g, p, y \rangle$ , where  $n$  is the problem size,  $g$  is the starting gate number,  $p$ , is a connection path, and  $y$  is the either the type of gate, if  $p = \text{self}$ , or  $y$  is the destination gate number,  $g(p)$ .*

**Definition 5.3** *A circuit family  $C$  of size  $Z(n)$  and depth  $T(n)$  is  $U_E$  uniform if there is a DTM recognizing  $L_{EC}$  in time  $\log Z(n)$ .  $C$  is  $U_{E^*}$  uniform if there is an ATM recognizing  $L_{EC}$  in time  $O(T(n))$  and space  $O(\log Z(n))$ .*

**Theorem 5.1** *If a problem has  $U_{E^*}$  uniform size and depth complexity  $Z(n)$  and  $T(n)$ , then it has  $U_E$  uniform size and depth complexity  $Z(n)^{O(1)}$  and  $O(T(n))$ .*

**Example 5.1** : Once again, consider the simple tree circuit which ANDs together  $n$  binary values. In section 2.3, we showed that this circuit is  $U_B$  uniform. We now show that this circuit is also  $U_E$  uniform, ie. it's local connections can be generated by an alternating Turing machine in  $O(\log n)$  time and space.

We start off by labeling nodes. We label the inputs nodes as  $I(1), \dots, I(n)$  and the output node as  $O$ . We label the root node as 1. If a node is labeled  $i$ , then we label its left child as  $2i$  and its right child as  $2i + 1$ . We can then recognize whether

$\langle n, g, p, y \rangle \in L_{EC}$  with the following program. Let  $\text{len}(p)$  be the number of links in  $p$ , and  $\text{mem}(p, i)$  be the  $i$ th link in  $p$ .

1. Check if input is asking for the gate type: if  $p = \text{self}$  then accept the input only if  $y = \text{AND}$ .
2. Calculate the node traced out by path  $p$ :

$$\text{node} = 2^{\text{len}(p)}g + \sum_{i < \text{len}(p)} 2^{\text{len}(p)-i}g'(i)$$

where  $g'(i) = 1$  if  $\text{mem}(p, i) = R$ , and  $g'(i) = 0$ , otherwise.

3. If  $\text{node} \geq 2n - 1$ , then reject because the path  $p$  is too long.
4. If  $\text{node} \geq n - 1$ , then accept only if  $y = I(\text{node} - n + 1)$ , since  $y$  is an input node.
5. Accept if  $y = \text{node}$ .

The program works by treating the path as a binary representation for part of the label of the destination node. The remaining part is filled in by the originating node. The program requires only  $O(\log n)$  space because it only uses three variables,  $\text{node}$  and  $i$ , which are each  $O(\log n)$  bits. Also, it executes in  $O(\log n)$  time. Although step (2) seems to require us to multiply and add  $n$  times, notice that it only requires the addition of powers of 2. Therefore, we do not need to use addition and multiplication algorithms, but can execute step (2) merely by collating bits derived from  $p$ . This can be done in  $O(\log n)$  time. Hence, we have shown that our tree-circuit is  $U_E$  uniform.

**Example 5.2 :** Now, consider a tree circuit which aggregates the value of  $n$  inputs  $\log n$  at a time. We will assume we have blackboxes (which contain  $NC^1$  circuits) that aggregate  $O(\log n)$  values at a time. Because each blackbox only accepts  $O(\log n)$  input values, it has  $O(\log \log n)$  depth, and its interconnection language  $L'_{EC}$  is computable in  $O(\log \log n)$  time. In this example, we will show that the combined tree structure is also  $U_E$  uniform. Note that a very similar circuit will be used later in section 5.2 to show that discrete log constants can be computed within  $U_E$  uniformity restrictions. At that time, we will provide the internal structure of this blackbox.

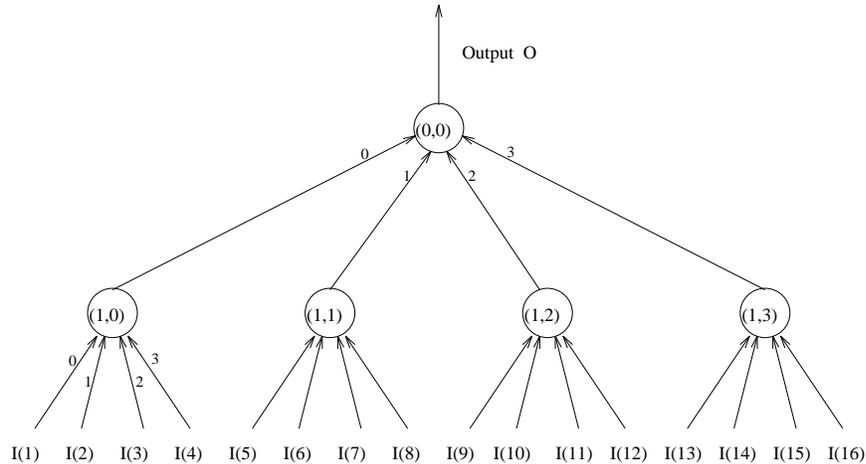


Figure 8: A  $\log n$ -ary tree AND circuit for  $n = 16$ .

Let  $b$  be the branching factor, and  $node\_size$  be the size of each blackbox. We will always take  $b$  to be the next largest power of two greater than  $\lceil \log n \rceil$ . We do this because we need to multiply by  $b$  often and we need  $b$  to be a power of two in order to stay within our logarithmic time limit.

We now carefully label the nodes. We have to be careful because we want to compute the labels of destination nodes by collating bits from the path instead of multiplying. We label the input nodes as  $I(1), \dots, I(n)$  and the output node  $Out$ . We label all our blackboxes by a tuple,  $(l, n)$ , and our internal nodes (inside the blackboxes) by a 3-tuple,  $(l, n, i)$ .  $l$  denotes the level in the tree,  $n$  denotes the number of the blackbox in that level, and  $i$  denotes the number of the node inside a blackbox. We label the root blackbox as  $(0, 0)$  and the internal nodes in the root blackbox as  $(0, 0, 0), \dots, (0, 0, node\_size - 1)$ . If a blackbox is labeled  $(i_1, i_2)$  then its internal nodes are labeled by  $(i_1, i_2, 0), \dots, (i_1, i_2, node\_size - 1)$ . Also, its children are labeled by  $(i_1 + 1, b * i_2), \dots, (i_1 + 1, b * i_2 + j - 1)$ . This labeling scheme is illustrated in figure 8.

The essence of showing  $U_E$  uniformity is by answering the following type of question: does the path  $\{llrlll\}$  lead us to input  $I(2)$  if we start from the root  $(0, 0, 0)$ ? We answer this question in  $O(\log n)$  time and space as follows. Let `generate_node` be the function which recognizes the connection language of the blackbox's internal nodes.

1. In our example, let  $n = 16$ . Our branching factor is 4 and we have two levels in our tree.
2. We note  $p$  travels through all the internal nodes of the blackboxes. We do not need or want all this extraneous information. Instead, we need to cull a new path  $p'$  which only describes the path taken at the blackbox level. For example,  $p' = \{0, 1\}$  from the root would mean that starting from the root, we take the input branch 0 and then take input branch 1. This would lead us to input node I(2).
3. Using the power of non-determinism, we guess the direction taken at the junction of each node:  $p' = \{0, 1\}$ .
4. Next, we guess at subpaths in  $p$  which travel inside each blackbox. In this case, we guess the starting points of our subpaths to be  $l' = \{0, 3\}$ . Our two subpaths are  $\{lll\}$  and  $\{rlll\}$ . This means that we guess that  $\{lll\}$  leads us through the first blackbox in our path and  $\{rlll\}$  leads us through the second blackbox.

Note that both steps 3 and 4 take  $O(\log n)$  time because the numbers involved are so small, that the total number of bits that we are guessing is still  $O(\log n)$ . For instance, our culled path  $p'$  has  $O(\log n / \log \log n)$  entries (one for each level of the tree), with each entry of size  $O(\log \log n)$  bits (because the blackboxes only deal with  $O(\log n)$  inputs).

5. Now, we have to verify that our guesses are correct using `generate_node`, the function for verifying paths inside the blackboxes. First, we check if, inside a blackbox, taking the path  $\{lll\}$  gets you to input branch 0. Next we check if taking the path  $\{rlll\}$  gets you to input branch 1. If so, our culled path  $p' = \{0, 1\}$  is correct.

This step takes  $O(\log n)$  time, because we run `generate_node`  $O(\log n / \log \log n)$  times, with each time taking  $O(\log \log n)$  time.

6. Now that we have our culled path  $p'$ , we follow it at the blackbox level to its destination. Following our labeling scheme, we start at the root  $(0, 0)$ , take branch 0 to node  $(1, 0)$  and then take branch 1 to node I(2). This step takes

$O(\log \log n)$  time because we are only multiplying by  $b$  each time, and have selected  $b$  to be a power of 2.

7. Therefore, we have verified that  $\{lll rlll\}$  leads to node I(2).

A program that recognizes the connection language of the combined tree structure would operate in a similar fashion as in our above example. It would run in  $O(\log n)$  time and  $O(\log n)$  space since almost every stage works with  $O(\log \log n)$  bit numbers. The labeling scheme presented above is crucial to the program being able run in logarithmic space and time. In particular, the selection of constants to be powers of 2 is important so that we avoid computations and instead just shift and append bits together. Thus, we have shown that our  $\log n$ -ary combination tree circuit is  $U_E$  uniform. Just as in our discussion for  $U_B$  uniformity, we see that in  $U_E$  uniformity, we can use tree and array combinations of  $NC^1$  circuits and still maintain uniformity.

As seen above, from the examples, the definition of  $U_E$  uniformity, by itself, is somewhat unwieldy for our purpose. The following theorem from [11], helps make the concept more adapted to our use.

Let  $ASPACE, TIME(S, T)$  represent problems which have ATM space complexity of  $S$  and time complexity of  $T$ . Let  $U_E\text{-SIZE,DEPTH}(S, T)$  represent problems with  $U_E$  uniform circuit space and time complexity of  $S$  and  $T$ .

**Theorem 5.2** For  $S(n), T(n) = \Omega(\log n)$ , and  $Z(n) = n^{\Omega(1)}$ ,

$$ASPACE, TIME(S(n), T(n)) \subseteq U_E\text{-SIZE,DEPTH}(2^{O(S(n))}, T(n)),$$

$$U_E\text{-SIZE,DEPTH}(Z(n), T(n)) \subseteq ASPACE, TIME(\log Z(n), T(n)).$$

This theorem illustrates the close relationship between uniform circuits and ATM's. Using this theorem, we see that an  $NC$  circuit family is uniform if it can be generated by another uniform  $NC$  circuit family. Therefore, the definition has equalized the power between the circuit generator and the computations inside the circuit.

Since a uniform  $NC$  circuit family is quite powerful,  $U_E$  uniformity includes circuit elements like small arithmetic units, tables, selectors, and linear and tree combinations of them. As in the case of  $U_B$  uniformity, we are mainly concerned with the

precomputed constants in our tables. However, instead of showing all of them can be generated in logarithmic space, but polynomial time, as in  $U_B$  uniformity, here we have to show that each one can be generated in limited time and space. Using the above theorems, we can either show that they can be generated by a DTM in logarithmic time (strict  $U_E$  uniformity), in an ATM with both logarithmic space and time ( $U_{E^*}$  uniformity), or by a  $U_E$  uniform  $NC^1$  circuit (theorem 5.2).

Finally, to relate  $U_E$  uniformity back to our old  $U_B$  uniformity.

**Theorem 5.3** *For a circuit family with time complexity  $T(n) > O(\log^2 n)$ ,  $U_B$  uniformity  $\Rightarrow U_{E^*}$  uniformity. For any problem,  $U_E$  uniformity  $\Rightarrow U_B$  uniformity.*

## 5.2 $U_E$ Uniform Division

In showing  $U_E$  uniformity, we are most concerned with the precomputed constants, since the rest of the circuit is just array and tree combinations of uniform subcircuits. There are two main tables of pre-computed constants in our circuit, the table of primes and the table of discrete logs. Since it is not intuitively obvious how to generate the  $i$ th prime, nor how to compute a discrete log in logarithmic time and space, we will describe  $U_E$  uniform subcircuits to generate these constants. By theorem 5.2, we know that these constants can be also generated by an ATM in  $O(\log n)$  time and space, hence showing the whole division circuit to be  $U_E$  uniform.

**Theorem 5.4** *Given  $k = O(n/\log n)$ , we can generate the  $k$ th prime in  $O(\log n)$  time using a  $U_E$  uniform circuit.*

**Proof :**

1. Estimate the number of integers,  $C$ , we have to test before we find the necessary number of primes. We know that  $C = O(n)$ .
2. For each  $i < C$ ,  $j < C$ , compute whether  $i$  divides  $j$ :

$$\text{DIVIDES}[i, j] = (i \bmod j = 1).$$

3. For each  $i < C$ , compute whether  $i$  is prime:

$$\text{PRIMES}[i] = \neg\left(\bigvee_{j < C} \text{DIVIDES}[i, j]\right).$$

4. For each  $i < C$ , compute  $i$ 's position in our table of primes:

$$\text{POSITION}[i] = \sum_{j < i} \text{PRIMES}[j].$$

5. Find  $i$  such that  $\text{POSITION}[i] = k$ .

This circuit has a depth of  $O(\log n)$  since the steps are either  $O(\log n)$  bit arithmetic or  $O(\log n)$  depth tree reductions. Similarly, the circuit is  $U_E$  uniform since each step is  $U_E$  uniform.  $\square$

**Theorem 5.5** *Given a modulus  $m < n$ , and integer  $h < m$ , we can compute the discrete log  $l$  of  $h$  in  $O(\log n)$  time using a  $U_E$  uniform circuit.*

**Proof :** The hard part here is computing  $i^j \bmod m$  in  $O(\log n)$  time. We cannot use the standard recursive technique of multiplying pairs. Under the standard technique, we need  $O(\log n)$  recursive steps for multiplying  $O(n)$  numbers, with each step requiring at least  $O(\log \log n)$  time to compute  $O(\log n)$  bit arithmetic. This takes a total of  $O(\log n \log \log n)$ , which is too slow.

For the moment, suppose we had an  $O(\log n)$  depth,  $U_E$  uniform subcircuit to compute  $i^j \bmod m$ . Then, we can compute the discrete log  $l$  in  $O(\log n)$  time with the following  $U_E$  uniform selector circuit. The selector circuit is  $U_E$  uniform because it is just array and tree combinations of  $U_E$  uniform subcircuits.

1. For each  $i < m$ ,  $j < m$ , compute

$$\text{POWER}[i, j] = i^j \bmod m.$$

2. For each  $i < m$ , compute

$$\text{GENERATOR}[i] = \bigvee_{j < m} (\text{POWER}[i, j] \neq i).$$

3. Compute  $g$  to be the first  $i$  where  $\text{GENERATOR}[i] = 1$ .
4. Find  $l$ , where  $h = \text{POWER}[g, l]$ .

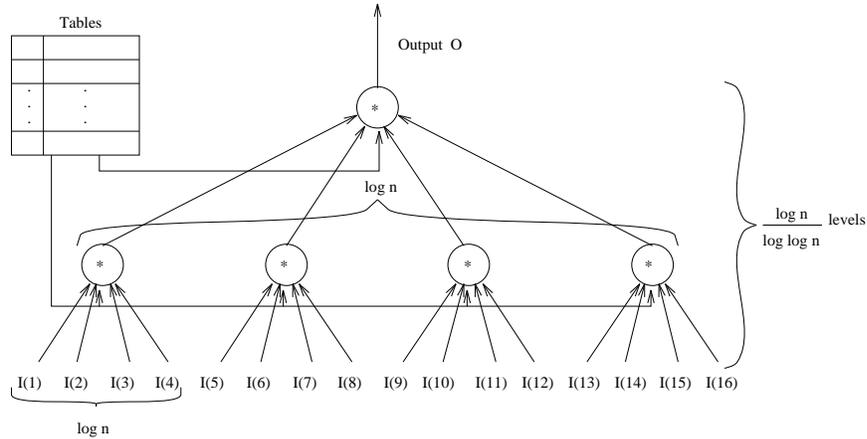


Figure 9: A circuit for computing  $i^j \bmod m$ . Total time =  $\frac{\log n}{\log \log n} \cdot \log \log n = \log n$ .

To compute  $i^j \bmod m$  we think of it as  $(i^{\log n})^{j/\log n} \bmod m$ . Computation of  $i^{\log n} \bmod m$  can be done in  $O(\log \log n)$  time using the product circuit described in sections 3.1 and 4.3. We recursively apply this process until we get  $i^j \bmod m$ . Each recursive step takes an  $O(\log n)$  bit integer, raises it to the  $\log n$ -th power, then takes it mod  $m$ . Iterating this process  $O(\log n / \log \log n)$  times, we get  $i^j \bmod m$ . This requires a total of  $O(\log \log n) \cdot O(\log n / \log \log n) = O(\log n)$  time.

We do not totally solve the problem by merely recursively applying the product circuit, however, because the product circuit itself uses precomputed constants. Specifically, the iterated product circuit also uses a table of discrete logs because it is based on the same techniques as our division circuit. Fortunately, we have reduced the size of the precomputed constants. In particular, our new table of discrete logs is reduced to  $O(\log \log n)$  bit integers raised to the  $\log n \log n$ -th power, because the product circuits operate on  $O(\log n)$  bit integers. Just as our  $n$ -bit division circuit required us to precompute  $O(\log n)$  bit constants, our  $O(\log n)$  bit iterated product circuit requires us to precompute  $O(\log n \log n)$  bit constants.

This is a much easier problem because our time and space constraints are still  $O(\log n)$  despite our problem size being reduced to  $O(\log \log n)$ . This is because these tables need only be computed once before the recursive process begins. We can compute the  $i^j \bmod m$  entries in our  $O(\log \log n)$  bit discrete log tables in  $O(\log n)$  time and space by sequentially multiplying  $i, j$  times. This completes our recursive proof. The whole circuit is  $U_E$  uniform because the small tables, the small product circuit, the  $\log n$ -ary collating tree (see example 5.2) and the selector circuit can all

be constructed by an ATM in  $O(\log n)$  time and space.

Therefore, we have shown that the discrete log of  $m$  can be computed in  $O(\log n)$  time using a  $U_E$  uniform circuit.  $\square$

Combining theorems 5.4 and 5.5, we have proved the following result.

**Theorem 5.6** *The division problem can be solved by a  $U_E$  uniform circuit family in  $O(\log n)$  depth. Therefore, the division, powering, and iterated products problems are all in  $NC^1$ .*

## Chapter 6

### Conclusion

#### 6.1 Summary

In this thesis, we have shown that the problem of integer division is in  $NC^1$ . The problem of integer division is to compute  $\lfloor x/y \rfloor$  for binary integers  $x, y$ . There is no known  $O(\log n)$  circuit that uses purely binary arithmetic to compute division. Instead, our circuit mainly uses arithmetic in the Chinese remainder representation (CRR) system. Chinese remainder representation is good for parallel arithmetic because numbers are represented as small, independent units. This is opposed to regular binary arithmetic circuits, where parallelism is limited by the “carry”.

We first presented circuits for basic arithmetic operations in CRR such as addition, multiplication, rank computation, modulo and comparison. These basic tools allowed us to describe a circuit for CRR division, which is the same as integer division except that the inputs and outputs are all in CRR. The CRR division circuit computed an  $n$ -bit underapproximation to the reciprocal using a geometric series. It took advantage of the fact that multiplying  $n$  CRR integers can be computed in only  $O(\log n)$  depth. Finally, we described a circuit for converting the result back to binary, based on the CRR division circuit itself. Together, this forms an  $NC^1$  circuit for integer division.

In order to simplify our discussion, we first showed that our division circuit is in  $NC^1$  under the more relaxed  $U_B$  definition of uniformity. Building on this result, we then established that the division circuit is also uniform under the more restrictive  $U_E$  uniformity condition.

## 6.2 Further Research

The issue of the practicality of  $NC^1$  division is still open. The division circuits described here are not optimized for up to polynomial factors in size and constant factors in depth. Instead, they are formulated in a way so that they are more easily seen to be in  $NC^1$ . Although this circuit has good asymptotic behavior, the constant factors make it unwieldy even for large number computations used in cryptographic systems. It would be interesting to see this circuit simplified so that it is competitive with traditional binary division circuits for large number computations.

On the theoretical side, it would be useful to see an explicit log-space algorithm for iterated product. This problem had hypothesized as a candidate for a language in PTIME - log-space.

## Bibliography

- [1] P. Beame, S. Cook, and H. Hoover. Log depth circuits for division and related problems. *SIAM Journal of Computing*, 15:994–1003, 1986.
- [2] A. Borodin. On relating time and space to size and depth. *SIAM Journal of Computing*, 6:733–744, 1977.
- [3] G. Davida and B. Litow. Fast parallel arithmetic via modular representation. *SIAM Journal of Computing*, 20:756–765, 1991.
- [4] G. Dimauro, S. Impedovo, and G. Pirlo. A new technique for fast number comparison in the residue number system. *IEEE Transactions on Computers*, 42(5):608–612, 1993.
- [5] J. Hennessey and D. Patterson. *Computer Architecture, A Quantitative Approach*. Morgan Kaufman Publishing, Inc. Santa Mateo CA, 1990.
- [6] R. Karp and V. Ramachandran. A survey of parallel algorithms for shared memory machines. Technical Report UCB/CSD 88/408, EECS, U of California, Berkeley, 1988.
- [7] F. T. Leighton. *Introduction to Parallel Algorithms & Architectures: Arrays, Trees and Hypercubes*. Morgan Kaufman Publishing, Santa Mateo CA, 1992.
- [8] M. Lu and J. S. Chiang. A novel division algorithm for the residue number system. *IEEE Transactions on Computers*, 41(8):1026–1032, 1992.
- [9] I. Niven and H. Zuckerman. *An Introduction to the Theory of Numbers*. John Wiley & Sons, NY, 1980.
- [10] J. Reif. Logarithmic depth circuits for algebraic functions. *SIAM Journal of Computing*, 15:231–242, 1986.

- [11] W. Ruzzo. On uniform circuit complexity. *Journal of Computer and System Sciences*, 22:365–383, 1981.
- [12] N. Shankar and V. Ramachandran. Efficient parallel circuits and algorithms for division. Technical Report ACT 79, Coordinated Science Lab, U of Illinois, 1987.
- [13] R. Tanaka and N. Szabo. *Residue Arithmetic and its Application to Computer Technology*. McGraw-Hill, 1968.
- [14] T. V. Vu. Efficient implementation of the chinese remainder theorem for sign detection and residue decoding. *IEEE Transactions on Computing*, 34(7):646–651, 1985.