Towards Optimal Simulations of Formulas by Bounded-Width Programs[1]

Richard Cleve[2]

TR-90-013

March 9, 1990

Abstract

We show that, over an arbitrary ring, for any fixed $\varepsilon > 0$, all balanced algebraic formulas of size s are computed by algebraic straight-line programs that employ a constant number of registers and have length $O(s^{1+\varepsilon})$. In particular, in the special case where the ring is $GF(2)$, we obtain a technique for simulating balanced Boolean formulas of size s by bounded-width branching programs of length $O(s^{1+\varepsilon})$, for any fixed $\varepsilon > 0$. This is an asymptotic improvement in efficiency over previous simulations in both the Boolean and algebraic setting.

---

[1] To appear in Proc. 22nd Ann. ACM Symp. on Theory of Computing, 1990.

[2] International Computer Science Institute, 1947 Center Street, Berkeley, CA 94704.

1 Introduction

The first investigation of the computational power of programs whose storage capacity is limited to a constant number of items was made by Borodin *et al.* (1983), and Chandra, *et al.* (1983). These authors considered the polynomial-length bounded-width branching program as a model of computation for functions from $\{0,1\}^n$ to $\{0,1\}$. These programs are equivalent to polynomial-length straight-line programs that employ a constant number of $\{0,1\}$-valued read/write registers and have read-only access to their inputs.

It had been conjectured that the polynomial-length bounded-width branching program was a weaker model of computation than the polynomial-size formula (and this conjecture was partially supported by the results of Borodin *et al.* (1983), Chandra *et al.* (1983), Yao (1983), Púdlak (1984), and Ajtai *et al.* (1986)). Then Barrington (1986), using a beautiful construction, proved that in fact the two models of computation are equivalent. Ben-Or and Cleve (1988) extended Barrington's result to an algebraic setting by showing that, over an arbitrary ring, the functions computed by polynomial-size *algebraic* formulas are also computed by polynomial-length *algebraic* straight-line programs that use only a constant number of registers. This result generalizes Barrington's since, in the special case where the ring is $GF(2)$, it implies the former result.

A feature of the constructions of Barrington (1986) as well as those of Ben-Or and Cleve (1988) is that, in general, they incur a polynomial increase in the total amount of work required to perform the computation: if the original formulas have size s and are balanced (i.e. have depth bounded by $\lceil \log s \rceil$) then the resulting straight-line programs may have length up to $\Omega(s^2)$. (And when the formulas are not balanced, the resulting straight-line programs may have length even more than quadratic in s.) Cai and Lipton (1989) showed an alternative construction for the Boolean case that is subquadratic for balanced formulas. Their construction translates balanced Boolean formulas of size s into bounded-width branching programs of length $O(s^{1.811\cdots})$.

We show that (in the algebraic as well as the Boolean setting) the exponent in the simulation can be made arbitrarily close to one. That is, given any fixed $\epsilon > 0$, we show that all (algebraic or Boolean) balanced formulas of size s are computed by straight-line programs that use a constant number of registers and have length $O(s^{1+\epsilon})$. This affirms a conjecture of Cai and Lipton (1989), whose $O(s^{1.811\cdots})$ construction is the most efficient previously known one for the Boolean case. In the algebraic case, the most efficient previously known construction is that of Ben-Or and Cleve (1988), which is $O(s^2)$. Also, for our $O(s^{1+\epsilon})$ construction, the formulas only need to be "approximately balanced" in the sense that their depth is bounded by $\lceil (1+\delta)\log s \rceil$ for some $\delta < \epsilon$.

In our constructions, the number of registers required to attain a particular bound on the exponent increases as the exponent approaches 1. More specifically, to achieve an exponent of $1+\epsilon$, the number of registers in our construction must be $4^{(\frac{1}{\epsilon}+1)} - 1$. For this reason, it is important to regard the value of ϵ as fixed. In practical terms, ϵ cannot be made too small without a huge number of registers.

It is of interest that the straight-line programs that arise in our constructions have a special form: they consist of statements that apply (special) invertible linear operations to the registers. More precisely, if the number of registers is w, and one regards each possible

configuration of values of the registers as a vector in $\mathcal{R}^w$ then the effect of executing a statement of these programs is equivalent to multiplying this vector by a $w \times w$ matrix with determinant 1 (and one entry of this matrix is an input or its negation, and the other entries are constants). Thus, the statements that constitute these programs can be viewed as elements of $SL_w(\mathcal{R})$, the *special linear group*, consisting of $w \times w$ matrices with determinant 1. (In Barrington's (1986), and Cai and Lipton's (1989) constructions, the statements of the programs can be viewed as elements of the group S_5, of permutations on a five-element set. To further compare our results, we note that, when the ring is $GF(2)$, our programs are, in the language of Barrington, "permutation branching programs of width $2^w - 1$" (where the "states" are the nonzero elements of $\{0,1\}^w$).)

The primary motivation for this research is complexity theoretic: to determine more precise relationships between two different characterizations of the complexity class NC^1. Also, there are a number of potential applications. Cai and Lipton (1989, p. 569) explain that computations expressed in terms of bounded-width branching programs, if they are efficient enough, are particularly well suited for hardware implementations by certain "reconfigurable" VLSI chips. Also, Kilian (1988, pp. 23–26) has shown how to use permutation branching programs to reduce the number of rounds of interaction required for certain cryptographic protocols. Roughly, by expressing formulas as bounded-width permutation branching programs, Kilian shows how to construct protocols that perform "oblivious function evaluations" of formulas in a constant number of rounds (whereas, previously proposed constructions require $\Omega(s)$ rounds, for formulas of size s). A possible drawback of Kilian's construction is that, in exchange for the reduction in the number of rounds of interaction, the total number of bits that the parties need to communicate increases. This increase is due to the cost incurred by simulating formulas by bounded-width permutation branching programs. Thus, more efficient simulations are of value here.

2 Models of Computation

Let $(\mathcal{R}, +, \cdot, 0, 1)$ be an arbitrary ring. In this section, we define formulas and straight-line programs over $\mathcal{R}$. Our definition of a formula is very standard, and our definition of a "linear bijection straight-line program" is compatible with the standard definition of a straight-line program (with additional restrictions).

Definition 1: A *formula* over $(\mathcal{R}, +, \cdot, 0, 1)$ of depth d is defined as follows. A depth 0 formula is either c, for some $c \in \mathcal{R}$ (a *constant*) or x_u, for some $u \in \{1, 2, ...\}$ (an *input*). For $d > 0$, a depth d formula is either $(f + g)$ or $(f \cdot g)$, where f and g are formulas of depth d_f and d_g (respectively) and $d = \max(d_f, d_g) + 1$. The *size* of a formula is defined as follows. A depth 0 formula has size 1, and if f and g have size s_f and s_g (respectively) then the sizes of $(f + g)$ and $(f \cdot g)$ are both $s_f + s_g + 1$. A formula is *read-once* if, for each input x_u, it contains the subformula x_u at most once. A formula computes a function from $\mathcal{R}^n$ to $\mathcal{R}$ is a natural way (where n is the number of distinct inputs occurring in the formula).

Definition 2: A *linear bijection straight-line program (LBS program)* over $(\mathcal{R}, +, \cdot, 0, 1)$ is a sequence of assignment statements of the form

$R_j \leftarrow R_j + (R_i \cdot c)$; or
$R_j \leftarrow R_j - (R_i \cdot c)$; or
$R_j \leftarrow R_j + (R_i \cdot x_u)$; or
$R_j \leftarrow R_j - (R_i \cdot x_u)$,

where $i, j \in \{1, ..., w\}$, $i \neq j$, $c \in \mathcal{R}$, and $u \in \{1, ..., n\}$. $R_1, ..., R_w$ are *registers*, and $x_1, ..., x_n$ are *inputs*. The *width* of an LBS program is w, the maximum number of registers that it uses. The *length* of an LBS program is the number of statements it contains. An LBS program is *read-once* if, for each $u \in \{1, ..., n\}$, the symbol x_u occurs in at most one statement of the program. More generally, for $r \in \{1, 2, ...\}$, we say that an LBS program is *read-r* if each input symbol occurs in at most r statements of the program. LBS programs compute functions from $\mathcal{R}^n$ to $\mathcal{R}$ in a natural way, provided that we have some fixed convention about the initial values of registers, and about which register's final value taken to be the output of the computation.

For each specific value of the inputs $x_1, ..., x_n$, each statement in an LBS program induces a transformation on the vector consisting of the values of the registers $(R_1, ..., R_w)$. This transformation can be represented by the matrix whose diagonal entries are 1, whose (i, j)-th entry is $\pm c$ or $\pm x_u$ (depending on which of the four basic forms the statement takes), and whose other entries are 0. Executing a statement is equivalent to multiplying $(R_1, ..., R_w)$ on the right by the corresponding matrix. For example, the statement $R_1 \leftarrow R_1 + (R_2 \cdot x_1)$ corresponds to the matrix

$$\begin{pmatrix} 1 & 0 & 0 \\ x_1 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

(when $w = 3$). In this manner, the statements in an LBS program correspond to elements of $SL_w(\mathcal{R})$, the *special linear group* consisting of $w \times w$ matrices with determinant 1. In particular, in the language of Valiant (1979), an LBS program that uses w registers can be viewed as a "P-projection" of an iterated product of $w \times w$ matrices.

3 Results

The main result of this paper is Theorem 2. Lemma 3 plays an important role in the proof of Theorem 2 and may be of independent interest.

Definition 3: For distinct $i, j \in \{1, ..., w\}$, we say that an LBS program *offsets* R_j *by* $+R_i \cdot f(x_1, ..., x_n)$ if it transforms the values of the registers as follows. R_j is incremented by the value of R_i times $f(x_1, ..., x_n)$ and (importantly) all other registers incur no net change (i.e. for all $k \neq j$, R_k has the same final value as its initial value). For example, the single statement $R_1 \leftarrow R_1 + (R_2 \cdot x_1)$ offsets R_1 by $+R_2 \cdot x_1$. Similarly, we say that an LBS program *offsets* R_j *by* $-R_i \cdot f(x_1, ..., x_n)$ if it *decrements* R_j by the value of R_i times $f(x_1, ..., x_n)$ and causes no net change in the values of all other registers. For example, the single statement $R_1 \leftarrow R_1 - (R_2 \cdot x_1)$ offsets R_1 by $-R_2 \cdot x_1$.

Note that to compute $f(x_1, ..., x_n)$ it is sufficient to construct an LBS program that offsets R_j by $+R_i \cdot f(x_1, ..., x_n)$ if one adopts an initialization convention where R_i is initially 1 and R_j is initially 0, and one adopts the convention that the value of R_j is the output of the computation.

For convenience, we say that we have LBS programs that offset R_j by $\pm R_i \cdot f(x_1, ..., x_n)$ whenever we have an LBS program that offsets R_j by $+R_i \cdot f(x_1, ..., x_n)$ as well as an LBS program that offsets R_j by $-R_i \cdot f(x_1, ..., x_n)$.

We express the construction of Ben-Or and Cleve (1988) in our current formalism as follows.

Theorem 1 (Ben-Or and Cleve, 1988): *Over any ring $(\mathcal{R}, +, \cdot, 0, 1)$, any formula $f(x_1, ..., x_n)$ of depth d is computed by an LBS program of width three and length at most $(2^d)^2$.*

Proof: Recursively on d, the depth of $f(x_1, ..., x_n)$, we construct LBS programs that offset R_j by $\pm R_i \cdot f(x_1, ..., x_n)$ (for distinct $i, j \in \{1, 2, 3\}$).

The construction is trivial when $d = 0$: the appropriate single statement offsets R_j by $\pm R_i \cdot c$, or by $\pm R_i \cdot x_k$.

Suppose that we have LBS programs that offset R_j by $\pm R_i \cdot f(x_1, ..., x_n)$, and that offset R_j by $\pm R_i \cdot g(x_1, ..., x_n)$. Then we can offset R_j by $+R_i \cdot (f + g)(x_1, ..., x_n)$ by the LBS program

offset R_j by $+R_i \cdot f(x_1, ..., x_n)$
offset R_j by $+R_i \cdot g(x_1, ..., x_n)$,

and we can construct a similar program that offsets R_j by $-R_i \cdot (f + g)(x_1, ..., x_n)$. Less obviously, we can offset R_k by $+R_i \cdot (f \cdot g)(x_1, ..., x_n)$ by the LBS program

offset R_k by $-R_j \cdot g(x_1, ..., x_n)$
offset R_j by $+R_i \cdot f(x_1, ..., x_n)$
offset R_k by $+R_j \cdot g(x_1, ..., x_n)$
offset R_j by $-R_i \cdot f(x_1, ..., x_n)$.

One can verify that this program has the required properties by the identities

$$r_j + r_i \cdot f - r_i \cdot f = r_j$$

and

$$r_k - r_j \cdot g + (r_j + r_i \cdot f) \cdot g = r_k + r_i \cdot (f \cdot g) \ .$$

Also, there is a similar construction that offsets R_k by $-R_i \cdot (f \cdot g)(x_1, ..., x_n)$.

Since the maximum recursive factor per level of depth in this construction is four, if the depth of $f(x_1, ..., x_n)$ is d, the resulting LBS program has length at most $4^d = (2^d)^2$. □

Cai and Lipton (1989) were the first to observe that more efficient simulations of formulas by bounded-width programs—where the exponent of 2^d is less than 2—are possible. Their idea is to construct the simulations recursively, but rather than with respect to single gates,

with respect to clusters consisting of several gates. Our construction in Theorem 2 extends this idea. Informally, for any fixed k, we present a construction that is recursive with respect to clusters of gates corresponding to intermediate subformulas of depth k and size 2^k. We shall show that the recursive factor of our construction is $4 \cdot 2^k$ (whereas, the total recursive factor that results from a repeated application of the construction in Theorem 1 is, in general, $(2^k)^2$).

Theorem 2: *Over any ring* $(\mathcal{R}, +, \cdot, 0, 1)$, *for any fixed* k, *every formula* $f(x_1, ..., x_n)$ *of depth* d *is simulated by an LBS program of width* $2^{k+2} - 1$ *and length* $O((2^d)^{(1+\frac{2}{k})})$.

The proof of Theorem 2 (which appears at the end of this section) is an application of Lemma 3 and Lemma 4 (both are stated and proven below).

A formula $f(x_1, ..., x_n)$ of depth $d+k$ can be represented in terms of a balanced read-once formula $g(y_1, ..., y_{2^k})$ of depth k, and 2^k subformulas $h_u(x_1, ..., x_n)$ $(u \in \{1, ..., 2^k\})$ of depth bounded by d such that

$$f(x_1, ..., x_n) = g(h_1(x_1, ..., x_n), ..., h_{2^k}(x_1, ..., x_n)).$$

In order to simulate $f(x_1, ..., x_n)$ efficiently in terms of $h_u(x_1, ..., x_n)$ $(u \in \{1, ..., 2^k\})$, we show (in Lemma 3) that $g(y_1, ..., y_{2^k})$ has an alternate representation as a special kind of straight-line program, called a restricted multiplication straight-line program (defined below). Then, using this alternate representation, we show (in Lemma 4) how to carry out the efficient k-level simulation, and, finally, we apply this to prove Theorem 2.

Basically, restricted multiplication straight-line programs (defined formally below) differ from conventional straight-line programs in that they do not allow the product of the contents of two registers to be computed explicitly. More precisely, statements of the form $Q_i \leftarrow Q_j \cdot Q_k$, where Q_i, Q_j, Q_k are registers are disallowed. Only products of registers and constants $(Q_i \leftarrow Q_j \cdot c)$ or registers and inputs $(Q_i \leftarrow Q_j \cdot y_u)$ may be computed explicitly.

Definition 4: Define a *restricted multiplication straight-line program (RMS program)* as a sequence of assignment statements of the form

$Q_i \leftarrow 1$; or
$Q_i \leftarrow Q_j$; or
$Q_i \leftarrow Q_j + Q_k$; or
$Q_i \leftarrow Q_j \cdot c$; or
$Q_i \leftarrow Q_j \cdot y_u$,

where $i, j, k \in \{1, ..., w\}$, $c \in \mathcal{R}$, and $u \in \{1, ..., m\}$. Here, in order make the distinction between RMS programs and LBS programs clear, we denote the *registers* as $Q_1, ..., Q_w$, and the *inputs* as $y_1, ..., y_m$. Naturally, the *width* of an RMS program is w, and its *length* is the number of statements it contains. An RMS program is *read-once* if, for each $u \in \{1, ..., m\}$, the symbol y_u occurs in at most one statement of the program.

We do not use an initialization convention in order to unambiguously associate functions with RMS programs; rather, we restrict ourselves to RMS programs that are *unambiguous* with respect to all of their registers in the following sense. A RMS program is *unambiguous*

with respect to register Q_i if all assignments of values to registers made in the program do not depend on the initial value of register Q_i. (For example the RMS program consisting of $Q_2 \leftarrow Q_1 \cdot y_1$ followed by $Q_1 \leftarrow 1$ is ambiguous with respect to Q_1, but, if the order of the two statements is reversed, the RMS program is unambiguous with respect to both of its registers.) We say that an RMS program *computes* a function from $\mathcal{R}^n$ to $\mathcal{R}$ if it is unambiguous with respect to all of its registers, and some fixed register's final value is the value of the function.

It is well known that formulas of size s and depth d are computed by (unrestricted) straight-line programs of length s that use $d+1$ registers. The straight-line programs simply evaluate all the nodes of the formulas in a "depth-first traversal" order. In general, such an evaluation requires assignment statements of the form $Q_i \leftarrow Q_j \cdot Q_k$ (to evaluate multiplication nodes whose subformulas are not constants or inputs). Also, in general, disallowing assignment statements of the form $Q_i \leftarrow Q_j \cdot Q_k$ from straight-line programs results in a strictly weaker model of computation (since all RMS programs of polynomial-length compute only functions of polynomial degree, whereas polynomial-length *unrestricted* straight-line programs compute some functions of exponential degree (e.g. by repeated squaring)). Nevertheless, Lemma 3 implies that disallowing these multiplication statements from straight-line programs does not significantly reduce their efficiency in evaluating formulas. It should be noted that, although it may be of independent interest, the fact that the RMS program in Lemma 3 has width bounded by $d+1$ is not used for the proof of Theorem 2.

Lemma 3: *Let $g(y_1, ..., y_m)$ be a read-once formula of size s and depth d. Then there is a read-once RMS program of length at most $2s$ and width at most $d+1$ that computes $g(y_1, ..., y_m)$. (Moreover, as a technical convenience for Lemma 4, the first statement in the RMS program is $Q_1 \leftarrow 1$ and this is the only statement of the form $Q_i \leftarrow 1$ in the program.)*

Proof: For a formula $g(y_1, ..., y_m)$, we say that an RMS program $g(y_1, ..., y_m)$-*scales* Q_i if it assigns to register Q_i the value of its initial contents multiplied by $g(y_1, ..., y_m)$ and is unambiguous with respect to every register except Q_i. (Note that here we are not concerned about the effect of the program on its other registers besides Q_i as long as this effect is unambiguous.) For example, the single statement program $Q_1 \leftarrow Q_1 \cdot y_1$, y_1-scales Q_1.

To compute $g(y_1, ..., y_m)$, it is sufficient to construct a program that $g(y_1, ..., y_m)$-scales Q_1 because then, by preceding this program by the statement $Q_1 \leftarrow 1$, the required program is obtained.

Inductively on d, the depth of $g(y_1, ..., y_m)$, we construct RMS programs that $g(y_1, ..., y_m)$-scale Q_1.

When $d = 0$, the single statements $Q_1 \leftarrow Q_1 \cdot c$ and $Q_1 \leftarrow Q_1 \cdot y_u$ trivially c-scale Q_1 and y_u-scale Q_1 (respectively) in width 1.

Suppose that we have RMS programs of width w that $f(y_1, ..., y_m)$-scale Q_1 and $g(y_1, ..., y_m)$-scale Q_1.

Then we can $(f \cdot g)(y_1, ..., y_m)$-scale Q_1 in width w by the program

$f(y_1, ..., y_m)$-scale Q_1
$g(y_1, ..., y_m)$-scale Q_1 .

Also, in width $w+1$, we can $(f+g)(y_1, ..., y_m)$-scale Q_1 as follows. First, modify the program that $f(y_1, ..., y_m)$-scales Q_1 so that every reference to register Q_1 is replaced by a reference to register Q_{w+1}. The resulting program $f(y_1, ..., y_m)$-scales Q_{w+1} and makes no reference to register Q_1. Then the program

$Q_{w+1} \leftarrow Q_1$
$f(y_1, ..., y_m)$-scale Q_{w+1}
$g(y_1, ..., y_m)$-scale Q_1
$Q_1 \leftarrow Q_1 + Q_{w+1}$

$(f+g)(y_1, ..., y_m)$-scales Q_1.

It is straightforward to verify that if $f(y_1, ..., y_m)$ is read-once, and has size and depth bounded by s and d (respectively) then this construction results in a read-once RMS program of length at most $2s$ and width at most $d+1$. □

Lemma 4: *If $g(y_1, ..., y_m)$ is computed by a read-once RMS program of length l whose first statement is $Q_1 \leftarrow 1$ and such that this is the only statement of the form $Q_i \leftarrow 1$ in the program then there are read-4 (explained in Definition 2) LBS programs whose length and width are bounded by $8l$ and $l+1$ (respectively) that offset R_j by $\pm R_i \cdot g(y_1, ..., y_m)$.*

Proof: We can modify the RMS program to be "time-stamped", so that, for each $i \in \{1, ..., l\}$, its i-th statement performs an assignment on the register Q_i, and so that the output of the program is the final contents of Q_l (the modified program has width l). Let $\mathcal{S}_i$ denote the i-th statement of this modified program, and let $f_i(y_1, ..., y_m)$ be the unique value that is assigned to register Q_i.

We construct two sequences of LBS programs, $\mathcal{P}_i^+$, $\mathcal{P}_i^-$ $(i \in \{1, ..., l\})$. Informally, for $i \in \{1, ..., l\}$, the effect of $\mathcal{P}_i^+$ is to "simultaneously offset" the registers $R_2, ..., R_{i+1}$ by the values $+R_1 \cdot f_1(y_1, ..., y_m), ..., +R_1 \cdot f_i(y_1, ..., y_m)$ (respectively). More precisely, the effect of executing $\mathcal{P}_i^+$ is equivalent to that of executing an LBS program of the form

offset R_2 by $+R_1 \cdot f_1(y_1, ..., y_m)$
offset R_3 by $+R_1 \cdot f_2(y_1, ..., y_m)$
$\vdots$
offset R_{i+1} by $+R_1 \cdot f_i(y_1, ..., y_m)$.

(And $\mathcal{P}_i^-$ are defined similarly to simultaneously offset the registers $R_2, ..., R_{i+1}$ by the values $-R_1 \cdot f_1(y_1, ..., y_m), ..., -R_1 \cdot f_i(y_1, ..., y_m)$, respectively.) Note that then

$\mathcal{P}_l^+$
$\mathcal{P}_{l-1}^-$

offsets R_{l+1} by $+R_1 \cdot f(y_1, ..., y_m)$ and, thus, is the required program.

Now, we construct $\mathcal{P}_i^+$ ($i \in \{1,...,l\}$) inductively on i as follows. Since the first statement in the RMS program is $Q_1 \leftarrow 1$, it follows that $f_1(y_1,...,y_m) = 1$, so $\mathcal{P}_1^+$ is constructed by the single statement

$$R_2 \leftarrow R_2 + (R_1 \cdot 1) \ .$$

Now, assume that $i > 1$ and $\mathcal{P}_{i-1}^+$ has been constructed. $\mathcal{S}_i$ is of four possible forms, which we treat as separate cases.

Case 1: $\mathcal{S}_i$ is of the form $Q_i \leftarrow Q_j$ (where $j < i$). In this case, $\mathcal{P}_i^+$ is

$$\begin{array}{l} R_{i+1} \leftarrow R_{i+1} - (R_{j+1} \cdot 1) \\ \mathcal{P}_{i-1}^+ \\ R_{i+1} \leftarrow R_{i+1} + (R_{j+1} \cdot 1) \ . \end{array}$$

Here, the identity

$$r_{i+1} - r_{j+1} \cdot 1 + (r_{j+1} + r_1 \cdot f_j) \cdot 1 = r_{i+1} + r_1 \cdot f_j$$

verifies that the resulting program $\mathcal{P}_i^+$ has the desired properties.

Case 2: $\mathcal{S}_i$ is of the form $Q_i \leftarrow Q_j + Q_k$ (where $j, k < i$). In this case, $\mathcal{P}_i^+$ is

$$\begin{array}{l} R_{i+1} \leftarrow R_{i+1} - (R_{j+1} \cdot 1) \\ R_{i+1} \leftarrow R_{i+1} - (R_{k+1} \cdot 1) \\ \mathcal{P}_{i-1}^+ \\ R_{i+1} \leftarrow R_{i+1} + (R_{j+1} \cdot 1) \\ R_{i+1} \leftarrow R_{i+1} + (R_{k+1} \cdot 1) \ . \end{array}$$

In this case, the identity

$$r_{i+1} - r_{j+1} \cdot 1 - r_{k+1} \cdot 1 + (r_{j+1} + r_1 \cdot f_j) \cdot 1 + (r_{k+1} + r_1 \cdot f_k) \cdot 1 = r_{i+1} + r_1 \cdot (f_j + f_k)$$

verifies that the resulting program $\mathcal{P}_i^+$ has the required properties.

Case 3: $\mathcal{S}_i$ is of the form $Q_i \leftarrow Q_j \cdot c$ (where $j < i$ and $c \in \mathcal{R}$). In this case, $\mathcal{P}_i^+$ is

$$\begin{array}{l} R_{i+1} \leftarrow R_{i+1} - (R_{j+1} \cdot c) \\ \mathcal{P}_{i-1}^+ \\ R_{i+1} \leftarrow R_{i+1} + (R_{j+1} \cdot c) \ . \end{array}$$

One can verify that the resulting program $\mathcal{P}_i^+$ has the required properties from the identity

$$r_{i+1} - r_{j+1} \cdot c + (r_{j+1} + r_1 \cdot f_j) \cdot c = r_{i+1} + r_1 \cdot (f_j \cdot c) \ .$$

Case 4: $\mathcal{S}_i$ is of the form $Q_i \leftarrow Q_j \cdot y_u$ (where $j < i$ and $u \in \{1,...,m\}$). In this case, $\mathcal{P}_i^+$ is

$R_{i+1} \leftarrow R_{i+1} - (R_{j+1} \cdot y_u)$
$\mathcal{P}_{i-1}^{+}$
$R_{i+1} \leftarrow R_{i+1} + (R_{j+1} \cdot y_u)$.

In this case, the verification that the resulting program $\mathcal{P}_i^+$ has the required properties follows from the identity

$$r_{i+1} - r_{j+1} \cdot y_u + (r_{j+1} + r_1 \cdot f_j) \cdot y_u = r_{i+1} + r_1 \cdot (f_j \cdot y_u) .$$

This completes the construction of $\mathcal{P}_i^+$ ($i \in \{1, ..., l\}$). The construction of $\mathcal{P}_i^-$ ($i \in \{1, ..., l\}$) is similar.

Note that each symbol y_u occurs in at most 2 statements of $\mathcal{P}_l^+$ and of $\mathcal{P}_l^-$. Therefore, the final program

$\mathcal{P}_l^+$
$\mathcal{P}_{l-1}^-$

is read-4. Also, the length and width of the final program are bounded by $8l$ and $l + 1$ (respectively). □

Proof of Theorem 2: Fix k arbitrarily. Given $2^{k+2} - 1$ registers, we show how to construct LBS programs that offset R_j by $\pm R_i \cdot f(x_1, ..., x_n)$ and have length $O((2^d)^{(1+\frac{2}{k})})$, where d is the depth of $f(x_1, ..., x_n)$. Our construction is k-level recursive with respect to the depth of the formula.

Each depth $d + k$ formula $f(x_1, ..., x_n)$ can be expressed as

$$f(x_1, ..., x_n) = g(h_1(x_1, ..., x_n), ..., h_{2^k}(x_1, ..., x_n)),$$

where $g(y_1, ..., y_{2^k})$ is a balanced read-once formula of depth k, and $h_u(x_1, ..., x_n)$ ($u \in \{1, ..., 2^k\}$) are formulas of depth bounded by d.

Suppose that, for all distinct $i, j \in \{1, ..., 2^{k+2} - 1\}$, and all $u \in \{1, ..., 2^k\}$ we have LBS programs that offset R_j by $\pm R_i \cdot h_u(x_1, ..., x_n)$. We construct LBS programs that offset R_j by $\pm R_i \cdot f(x_1, ..., x_n)$ as follows. By Lemma 3, there exists a read-once RMS program of length bounded by $2 \cdot (2^{k+1} - 1) = 2^{k+2} - 2$ that computes $g(y_1, ..., y_{2^k})$. By Lemma 4, there exist width-$(2^{k+2} - 1)$ read-4 LBS programs whose lengths are bounded by $8 \cdot (2^{k+2} - 2) = 2^{k+5} - 16$ that offset R_j by $\pm R_i \cdot g(y_1, ..., y_{2^k})$. By replacing each statement of the form $R_i \leftarrow R_i \pm (R_j \cdot y_u)$ ($i, j \in \{1, ..., 2^{k+2} - 1\}$, $i \neq j$, and $u \in \{1, ..., 2^k\}$) in these programs by an LBS program that offsets R_i by $\pm R_j \cdot h_u(x_1, ..., x_n)$, we obtain LBS programs that offset R_j by $\pm R_i \cdot g(h_1(x_1, ..., x_n), ..., h_{2^k}(x_1, ..., x_n))$, or, equivalently, that offset R_j by $\pm R_i \cdot f(x_1, ..., x_n)$, as required.

By induction, if the depth of $f(x_1, ..., x_n)$ is d, the above construction results in an LBS program of length $O((4 \cdot 2^k)^{\frac{d}{k}}) = O((2^d)^{(1+\frac{2}{k})})$. □

By specializing the ring to $GF(2)$, Theorem 2 immediately implies that, for any fixed k, every Boolean formula of depth d over the basis $\{\wedge, \oplus, 1\}$ can be simulated by an LBS

program of width $2^{k+2}-1$ and length $O((2^d)^{(1+\frac{2}{k})})$. By observing that in the construction used in the proof of Theorem 2 only the size of $g(y_1,...,y_{2^k})$ matters (not the depth), we can also show the following.

Corollary 5: *For any fixed k, every Boolean formula $f(x_1,...,x_n)$ of depth d over the basis $\{\wedge,\vee,\oplus,\neg\}$ is simulated by an LBS program of width $2^{k+2}-1$ and length $O((2^d)^{(1+\frac{2}{k})})$ over the ring $GF(2)$.*

4 Open Problems

In our constructions, $\Omega(4^{\frac{1}{\epsilon}})$ registers are required in order to obtain an exponent of $1+\epsilon$. It would be interesting to determine whether or not such an exponent can be obtained with fewer registers, such as $(\frac{1}{\epsilon})^{O(1)}$.

As far as we know, a polynomial increase in size is incurred when a *general* formula is converted into a *balanced* formula. Perhaps our technique can nevertheless be extended to simulate general formulas of size s by bounded-width programs of length $O(s^{1+\epsilon})$.

A particularly interesting consequence of Barrington's (1986) result is that the *MAJORITY* function from $\{0,1\}^n$ to $\{0,1\}$ is computed by a bounded-width program of length polynomial in n. This can be taken to mean that one can "count" the number of 1s in a string of length n with constant on-line storage (rather than the $\log n$ on-line storage that one might expect). It would be interesting to construct bounded-width programs of length $O(n^{1+\epsilon})$ that compute the *MAJORITY* function. Our present result does not solve this since, as far as we know, there is no balanced formula of size $O(n^{1+\epsilon})$ (for arbitrarily small $\epsilon>0$) over the basis $\{\wedge,\vee,\oplus,\neg\}$ that computes *MAJORITY*. Perhaps a more direct construction is possible, and such an approach may also yield a deeper understanding of how one can "count" with bounded on-line storage.

5 Acknowledgment

I thank Jin-yi Cai for drawing my attention to an error in an earlier version of this paper.

References

Ajtai, M., L. Babai, P. Hajnal, J. Komlós, P. Púdlak, V. Rödl, E. Szemedréi, and G. Turán (1986), "Two Lower Bounds for Branching Programs", *Proc. 18th Ann. ACM Symp. on Theory of Computing*, pp. 30–38.

Barrington, D. A. (1986), "Bounded-Width Polynomial-Size Branching Programs Recognize Exactly Those Languages in NC^1", *Proc. 18th Ann. ACM Symp. on Theory of Computing*, pp. 1–5. Final Version in *J. Computer System Sci.* **38**, 1989, pp. 150–164.

Ben-Or, M., and R. Cleve (1988), "Computing Algebraic Formulas Using a Constant Number of Registers", *Proc. 20th Ann. ACM Symp. on Theory of Computing*, pp. 254–257.

Borodin, A., D. Dolev, F. Fich, and W. Paul (1983), "Bounds for Width Two Branching Programs", *Proc. 15th Ann. ACM Symp. on Theory of Computing*, pp. 87–93.

Cai, J., and R. J. Lipton (1989), "Subquadratic Simulations of Circuits by Branching Programs", *Proc. 30th Ann. IEEE Symp. on Foundations of Computer Sci.*, pp. 568–573.

Chandra, A., M. Furst, and R. J. Lipton (1983), "Multiparty Protocols", *Proc. 15th Ann. ACM Symp. on Theory of Computing*, pp. 94–99.

Kilian, J. (1988), "Founding Cryptography on Oblivious Transfer", *Proc. 20th Ann. ACM Symp. on Theory of Computing*, pp. 20–31.

Púdlak, P. (1984), "A Lower Bound on the Complexity of Branching Programs", *Proc. Conf. on Math. Foundations of Computer Sci.*, pp. 480–489.

Valiant, L. G. (1979), "Completeness Classes in Algebra", *Proc. 11th Ann. ACM Symp. on Theory of Computing*, pp. 249–261.

Yao, A. C. (1983), "Lower Bounds by Probabilistic Arguments", *Proc. 24th Ann. IEEE Symp. on Foundations of Computer Sci.*, pp. 420–428.