# Computing with and without Arbitrary Large Numbers

Michael Brand

Faculty of IT, Monash University
Clayton, VIC 3800, Australia
michael.brand@alumni.weizmann.ac.il

**Abstract.** In the study of random access machines (RAMs) it has been shown that the availability of an extra input integer, having no special properties other than being sufficiently large, is enough to reduce the computational complexity of some problems. However, this has only been shown so far for specific problems. We provide a characterization of the power of such extra inputs for general problems.

To do so, we first correct a classical result by Simon and Szegedy (1992) as well as one by Simon (1981). In the former we show mistakes in the proof and correct these by an entirely new construction, with no great change to the results. In the latter, the original proof direction stands with only minor modifications, but the new results are far stronger than those of Simon (1981).

In both cases, the new constructions provide the theoretical tools required to characterize the power of arbitrary large numbers.

**Keywords:** integer RAM, complexity, arbitrary large number.

## 1 Introduction

The Turing machine (TM), first introduced in [1], is undoubtedly the most familiar computational model. However, for algorithm analysis it often fails to adequately represent real-life complexities, for which reason the random access machine (RAM), closely resembling the intuitive notion of an idealized computer, has become the common choice in algorithm design. Ben-Amram and Galil [2] write "The RAM is intended to model what we are used to in conventional programming, idealized in order to be better accessible for theoretical study."

Here, "what we are used to in conventional programming" refers, among other things, to the ability to manipulate high-level objects by basic commands. However, this ability comes with some unexpected side effects. For example, one can consider a RAM that takes as an extra input an integer that has no special property other than being "large enough". Contrary to intuition, it has been shown that such arbitrary large numbers (ALNs) can lower problem time complexities. For example, [3] shows that the availability of ALNs lowers the arithmetic time complexity[1] of calculating $2^{2^x}$ from $\Theta(x)$ to $\Theta(\sqrt{x})$. However, all previous

[1] Arithmetic complexity is the computational complexity of a problem under the RAM$[+, \dot{-}, \times, \div]$ model, which is defined later on in this section.

T-H.H. Chan, L.C. Lau, and L. Trevisan (Eds.): TAMC 2013, LNCS 7876, pp. 181–192, 2013.

attempts to characterize the contribution of ALNs dealt with problem-specific methods of exploiting such inputs, whereas the present work gives, for the first time, a broad characterization of the scenarios in which arbitrary numbers do and those in which they do not increase computational power.

In order to present our results, we first redefine, briefly, the RAM model. (See [4] for a more formal introduction.)

Computations on RAMs are described by *programs*. RAM programs are sets of *commands*, each given a *label*. Without loss of generality, labels are taken to be consecutive integers. The bulk of RAM commands belong to one of two types. One type is an *assignment*. It is described by a triplet containing a $k$-ary operator, $k$ operands and a target. The other type is a *comparison*. It is given two operands and a comparison operator, and is equipped with labels to proceed to if the comparison is evaluated as either true or false. Other command-types include unconditional jumps and execution halt commands.

The execution model for RAM programs is as follows. The RAM is considered to have access to an infinite set of *registers*, each marked by a non-negative integer. The input to the program is given as the initial state of the first registers. The rest of the registers are initialized to 0. Program execution begins with the command labeled 1 and proceeds sequentially, except in comparisons (where execution proceeds according to the result of the comparison) and in jumps. When executing assignments, the $k$-ary operator is evaluated based on the values of the $k$ operands and the result is placed in the target register. The output of the program is the state of the first registers at program termination.

In order to discuss the computational power of RAMs, we consider only RAMs that are comparable in their input and output types to TMs. Namely, these will be the RAMs whose inputs and outputs both lie entirely in their first register. We compare these to TMs working on one-sided-infinite tapes over a binary alphabet, where "0" doubles as the blank. A RAM will be considered equivalent to a TM if, given as an input an integer whose binary encoding is the initial state of the TM's tape, the RAM halts with a non-zero output value if and only if the TM accepts on the input.

Furthermore, we assume, following e.g. [5], that all explicit constants used as operands in RAM programs belong to the set $\{0, 1\}$. This assumption does not make a material difference to the results, but it simplifies the presentation.

In this paper we deal with RAMs that use non-negative integers as their register contents. This is by far the most common choice. A RAM will be indicated by RAM[*op*], where *op* is the set of basic operations supported by the RAM. These basic operations are assumed to execute in a single unit of time. We use the syntax $f(n)$-RAM[*op*] to denote the set of problems solvable in $f(n)$ time by a RAM[*op*], where $n$ is the bit-length of the input. Replacing "RAM[*op*]" by "TM" indicates that the computational model used is a Turing machine.

Note that because registers only store non-negative integers, such operations as subtraction cannot be supported without tweaking. The customary solution is to replace subtraction by "natural subtraction", denoted "$\dot{-}$" and defined by $a \dot{-} b \stackrel{\text{def}}{=} \max(a - b, 0)$. We note that if the comparison operator "$\leq$"

(testing whether the first operand is less than or equal to the second operand) is not supported by the RAM directly, the comparison "$a \leq b$" can be simulated by the equivalent equality test "$a \dot{-} b = 0$". Testing for equality is always assumed to be supported.

By the same token, regular bitwise negation is not allowed, and $\neg a$ is tweaked to mean that the bits of $a$ are negated only up to and including its most significant "1" bit.

Operands to each operation can be explicit integer constants, the contents of explicitly named registers or the contents of registers whose numbers are specified by other registers. This last mode, which can also be used to define the target register, is known as "indirect addressing". In [6] it is proved that for the RAMs considered here indirect addressing has no effect. We therefore assume throughout that it is unavailable to the RAMs.

The following are two classical results regarding RAMs. Operations appearing in brackets within the operation list are optional, in the sense that the theorem holds both when the operation is part of *op* and when it is not.

**Theorem 1 ([7]).** *PTIME-RAM*$[+, [\dot{-}], [\times], \leftarrow, [\rightarrow], Bool] = PSPACE$

and

**Theorem 2 ([8]).** *PTIME-RAM*$[+, \dot{-}, /, \leftarrow, Bool; \leq] = ER$*, where ER is the set of problems solvable by Turing machines in*

$$\left.2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\right\} n \tag{1}$$

*time, where n is the length of the input.*

Here, "/" indicates exact division, which is the same as integer division (denoted "$\div$") but is only defined when the two operands divide exactly. The operations "$\leftarrow$" and "$\rightarrow$" indicate left shift ($a \leftarrow b = a \times 2^b$) and right shift ($a \rightarrow b = a \div 2^b$), respectively, and *Bool* is shorthand for the set of all bitwise Boolean functions.

In this paper, we show that while Theorem 1 is correct, its original proof is not. Theorem 2, on the other hand, despite being a classic result and one sometimes quoted verbatim (see, e.g., [9]), is, in fact, erroneous.

We re-prove the former here, and replace the latter by a stronger result, for the introduction of which we first require several definitions.

**Definition 1 (Expansion Limit).** *Let $M = M_{op}(t, inp)$ be the largest number that can appear in any register of a RAM*$[op]$ *working on inp as its input, during the course of its first t execution steps.*

*We define $EL_{op}(f(n))$ to be the maximum of $M_{op}(f(n), inp)$ over all values of inp for which $len(inp) \leq n$. This is the maximum number that can appear in any register of a RAM*$[op]$ *that was initialized by an input of length at most n, after $f(n)$ execution steps.*

*The subscript 'op' may be omitted if understood from the context.*

As a slight abuse of notation, we use $\mathrm{EL}(t)$ to be the maximum of $M_{op}(t, inp)$ over all *inp* of length at most $n$, when $n$ is understood from the context and $t$ is independent of $n$. (The following definition exemplifies this.)

**Definition 2 (RAM-Constructability).** *A set of operations op is* RAM-constructable *if the following two conditions are satisfied: (1) there exists a RAM program that, given inp and t as its inputs, with n being the length of inp, returns in $O(t)$ time a value no smaller than $EL_{op}(t)$, and (2) each operation in op is computable in $EL(O(l))$ space on a Turing machine, where l is the total length of all operands and of the result.*

Our results are as follows.

**Theorem 3.** *For a RAM-constructable $op \supseteq \{+, /, \leftarrow, Bool\}$ and any function $f(n)$,*

$$\begin{aligned} O(f(n))\text{-}RAM[op] &= EL_{op}(O(f(n)))\text{-}TM \\ &= N\text{-}EL_{op}(O(f(n)))\text{-}TM \\ &= EL_{op}(O(f(n)))\text{-}SPACE\text{-}TM \\ &= N\text{-}EL_{op}(O(f(n)))\text{-}SPACE\text{-}TM\ , \end{aligned} \tag{2}$$

*where the new notations refer to nondeterministic Turing machines, to space-bounded Turing machines and to nondeterministic space-bounded Turing machines, respectively.*

Among other things, this result implies for polynomial-time RAMs that their computational power is far greater than ER, as was previously believed.

The theoretical tools built for proving Theorem 3 and re-proving Theorem 1 then allow us to present the following new results regarding the power of arbitrary large numbers.

**Theorem 4.** *PTIME-ARAM*$[+, [\overset{\bullet}{-}], [\times], \leftarrow, [\rightarrow], Bool] = PSPACE$.

**Theorem 5.** *Any recursively enumerable (r.e.) set can be recognized in $O(1)$ time by an ARAM*$[+, /, \leftarrow, Bool]$.

Here, "ARAM" is the RAM model assisted by an arbitrary large number. Formally, we say that a set $S$ is computable by an ARAM[*op*] in $f(n)$ time if there exists a Boolean function $g(inp, x)$, computable in $f(n)$ time on a RAM[*op*], such that $inp \in S$ implies $g(inp, x) \neq 0$ for almost all $x$ (all but a finite number of $x$) whereas $inp \notin S$ implies $g(inp, x) = 0$ for almost all $x$. Here, $n$ conventionally denotes the bit length of the input, but other metrics are also applicable.

We see, therefore, that the availability of arbitrary numbers has no effect on the computational power of a RAM without division. However, for a RAM equipped with integer division, the boost in power is considerable, to the extent that any problem solvable by a Turing machine in any amount of time or space can be solved by an ARAM in O(1) time.

## 2 Models without Division

### 2.1 Errata on [7]

We begin with a definition.

**Definition 3 (Straight Line Program).** *A* Straight Line Program *(*SLP*), denoted SLP*[*op*]*, is a list of tuples,* $s_2, \ldots, s_n$*, where each* $s_i$ *is composed of an operator,* $s_i^{op} \in op$*, and* $k$ *integers,* $s_i^1, \ldots, s_i^k$*, all in the range* $0 \leq s_i^j < i$*, where* $k$ *is the number of operands taken by* $s_i^{op}$*. This list is to be interpreted as a set of computations, whose targets are* $v_0, \ldots, v_n$*, which are calculated as follows:* $v_0 = 0$*,* $v_1 = 1$*, and for each* $i > 1$*,* $v_i$ *is the result of evaluating the operator* $s_i^{op}$ *on the inputs* $v_{s_i^1}, \ldots, v_{s_i^k}$*. The output of an SLP is the value of* $v_n$*.*

A technique first formulated in a general form in [10] allows results on SLPs to be generalized to RAMs. Schönhage's theorem, as worded for the special case that interests us, is that if there exists a Turing machine, running on a polynomial-sized tape and in finite time, that takes an SLP[*op*] as input and halts in an accepting state if and only if $v_n$ is nonzero, then there also exists a TM running on a polynomial-sized tape that simulates a RAM[*op*]. This technique is used both in [7] and in our new proof.

The proof of [7] follows this scheme, and attempts to create such a Turing machine. In doing so, this TM stores monomial-based representations of certain powers of two. These are referred to by the paper as "monomials" but are, for our purposes, integers.

The main error in [7] begins with the definition of a relation, called "vicinity", between monomials, which is formulated as follows.

> We define an equivalence relation called *vicinity* between monomials. Let $M_1$ and $M_2$ be two monomials. Let $B$ be a given parameter. If $M_1/M_2 < 2^{2^B}$ [...], then $M_1$ is in the vicinity of $M_2$. The symmetric and transitive closure of this relation gives us the full vicinity relation. As it is an equivalence relation, we can talk about two monomials being in the same vicinity (in the same equivalence class).

It is unclear from the text whether the authors' original intention was to define this relation in a universal sense, as it applies to the set of all monomials (essentially, the set of all powers of two), or whether it is only defined over the set of monomials actually used by any given program. If the former is correct, any two monomials are necessarily in the same vicinity, because one can bridge the gap between them by monomials that are only a single order of magnitude apart. If the latter is correct, it is less clear what the final result is. The paper does not argue any claim that would characterize the symmetric and transitive closure in this case.

However, the paper does implicitly assume throughout that the vicinity relation, as originally defined (in the $M_1/M_2 < 2^{2^B}$ sense) is *its own* symmetric and transitive closure. This is used in the analysis by assuming for any $M_i$ and

$M_j$ which are in the same vicinity (in the equivalence relation sense) that they also satisfy $2^{-(2^B)} < M_i/M_j < 2^{2^B}$, i.e. they are in the same vicinity also in the restrictive sense.

Unfortunately, this claim is untrue. It is quite possible to construct an SLP that violates this assumption, and because the assumption is central to the entire algorithm, the proof does not hold.

We therefore provide here an alternate algorithm, significantly different from the original, that bypasses the entire "vicinity" issue.

### 2.2 Our New Construction

Our proof adapts techniques from two previous papers: [11] (which uses lazy evaluation to perform computations on operands that are too long to fit into a polynomial-sized tape) and [12] (which stores operands in a hierarchical format that notes only the positions of "interesting bits", these being bit positions whose values are different than those of the less significant bit directly preceding them). The former method is able to handle multiplication but not bit shifting and the latter the reverse. We prove Theorem 1 using a sequence of lemmas.

**Lemma 1.** *In an SLP*$[+, \dot{-}, \times, \leftarrow, \rightarrow, Bool]$*, the number of interesting bits in the output* $v_n$ *grows at most exponentially with* $n$*. There exists a Turing machine working in polynomial space that takes such an SLP as its input, and that outputs an exponential-sized set of descriptions of bit positions, where bit positions are described as functions of* $v_0, \ldots, v_{n-1}$*, such that the set is a superset of the interesting bit positions of* $v_n$*.*

The fact that the number of interesting bits grows only exponentially given this operation set was noted in [7]. Our proof follows the reasoning of the original paper.

*Proof.* Consider, for simplicity, the instruction set $op = \{+, \times, \leftarrow\}$. Suppose that we were to change the meaning of the operator "$\leftarrow$", so that, instead of calculating $a \leftarrow b = a \times 2^b$, its result would be $a \leftarrow b = aX$, where $X$ is a formal parameter, and a new formal parameter is generated every time the "$\leftarrow$" operator is used. The end result of the calculation will now no longer be an integer but rather a polynomial in the formal parameters. The following are some observations regarding this polynomial.

1. The number of formal parameters is at most $n$, the length of the SLP.
2. The power of each formal parameter is at most $2^{n-k}$, where $k$ is the step number in which the parameter was defined. (This exponent is at most doubled at each step in the SLP. Doubling may happen, for example, if the parameter is multiplied by itself.)
3. The sum of all multiplicative coefficients in the polynomial is at most $2^{2^{n-2}}$. (During multiplication, the sum of the product polynomial's coefficients is the product of the sums of the operands' coefficients. As such, this value can at most square itself at each operation. The maximal value it can attain at step 2 is 2.)

If we were to take each formal variable, $X$, that was created at an "$a \leftarrow b$" operation, and substitute in it the value $2^b$ (a substitution that [7] refers to as the "standard evaluation"), then the value of the polynomial will equal the value of the SLP's output. We claim that if $p$ is an interesting bit position, then there is some product of formal variables appearing as a monomial in the result polynomial such that its standard evaluation is $2^x$, and $p \geq x \geq p - 2^n$.

The claim is clearly true for $n = 0$ and $n = 1$. For $n > 1$, we will make the stronger claim $p \geq x \geq p - 2^{n-2} - 2$. To prove this, note that any monomial whose standard evaluation is greater than $2^p$ cannot influence the value of bit $p$ and cannot make it "interesting". On the other hand, if all remaining monomials are smaller than $p - 2^{n-2} - 2$, the total value that they carry within the polynomial is smaller than $2^{p-2^{n-2}-2}$ times the sum of their coefficients, hence smaller than $2^{p-2}$. Bits $p-1$ and $p$, however, are both zero. Therefore, $p$ is not an interesting bit.

We proved the claim for the restricted operation set $\{+, \times, \leftarrow\}$. Adding logical AND ("$\wedge$") and logical OR ("$\vee$") can clearly not change the fact that bits $p-1$ and $p$ are both zero, nor can it make the polynomial coefficients larger than $2^{2^{n-2}}$.

Incorporating "$\dot{-}$" and "$\neg$" into the operation set has a more interesting effect: the values of bit $p-1$ and $p$ can both become "1". This will still not make bit $p$ interesting, but it does require a small change in the argument. Instead of considering polynomials whose coefficients are between 0 and $2^{2^{n-2}}$, we can now consider polynomials whose coefficients are between $-2^{2^{n-2}}$ and $2^{2^{n-2}}$. This changes the original argument only slightly, in that we now need to argue that in taking the product over two polynomials the sum of the absolute values of the coefficients of the product is no greater than the product of the sums of the absolute values of the coefficients of the operands.

Similarly, adding "$\rightarrow$" into consideration, we no longer consider only formal variables of the form $a \leftarrow b = aX$ but also $a \rightarrow b = \lfloor aY \rfloor$, where the standard evaluation of $Y$ is $2^{-b}$ and $\lfloor \cdot \rfloor$ is treated as a bitwise Boolean operation (in the sense that, conceptually, it zeroes all bit positions that are "to the right of the decimal point" in the product).

We can therefore index the set of interesting bits by use of a tuple, as follows. If $i_1, \ldots, i_k$ are the set of steps for which $s_{i_j}^{op} \in \{\leftarrow, \rightarrow\}$, the tuple will contain one number between $-2^{n-i_j}$ and $2^{n-i_j}$ for each $1 \leq j \leq k$, to indicate the exponent of the formal parameter added at step $i_j$, and an additional $k+1$'th element, between 0 and $2^n$ to indicate a bit offset from this bit position.

Though this tuple may contain many non-interesting bits, or may describe a single bit position by many names, it is a description of a super-set of the interesting bits in polynomial space. □

We refer to the set of bit positions thus described as the *potentially-interesting* bits, or *po-bits*, of the SLP.

**Lemma 2.** *Let $\mathcal{O}$ be an Oracle that takes an $\mathcal{S} \in SLP[+, \dot{-}, \times, \leftarrow, \rightarrow, Bool]$ as input and outputs the descriptions of all its po-bits in order, from least-significant*

*to most-significant, without repetitions. There exists a TM working in polynomial space but with access to $\mathcal{O}$ that takes as inputs an $\mathcal{S}' \in SLP[+, \dot{-}, \times, \leftarrow, \rightarrow, Bool]$ and the description of a po-bit position, $i$, of $\mathcal{S}'$, and that outputs the $i$'th bit of the output of $\mathcal{S}'$.*

*Proof.* Given a way to iterate over the po-bits in order, the standard algorithms for most operations required work as expected. For example, addition can be performed bit-by-bit if the bits of the operands are not stored, but are, rather, calculated recursively whenever they are needed. The depth of the recursion required in this case is at most $n$.

The fact that iterating only over the po-bits, instead of over all bit positions, makes no difference to the results is exemplified in Fig. 1.

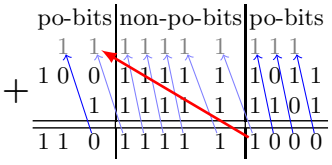


**Fig. 1.** An example of summing two numbers

As can be seen, not only are the non-po-bits all equal to the last po-bit preceding them, in addition, the carry bit going over from the last po-bit to the first non-po-bit is the same as the carry bit carried over from the last non-po-bit to the first po-bit. Because of this, the sequential carry bits across non-po-bits (depicted in light blue in Fig. 1) can be replaced by a single non-contiguous carry operation (the thick red arrow).

This logic works just as well for subtraction and Boolean operations. The only operation acting differently is multiplication. Implementing multiplication directly leads to incorrect results. Instead, we re-encode the operand bits in a way that reflects our original observation, that the operands can be taken to be polynomials with small coefficients in absolute value, though these coefficients may not necessarily be nonnegative.

The new encoding is as follows: going from least significant bit to most significant bit, a "0" bit is encoded as a 1 if preceded by a "1" and as 0, otherwise. A "1" bit is encoded as a 0 if preceded by a "1" and as $-1$, otherwise. It is easy to see that a number, $A$, encoded in regular binary notation but including a leading zero by a $\{0, 1\}$ sequence, $a_0, \ldots, a_k$, denoting coefficients of a power series $A = \sum_{i=0}^{k} a_i 2^i$, does not change its value if the $a_i$ are switched for the $b_i$ that are the result of the re-encoding procedure described. The main difference is that now the value of all non-po-bits is 0.

Proving that multiplication works correctly after re-encoding is done by observing its base cases and bilinear properties. The carry in the calculation is exponential in size, so can be stored using a polynomial number of bits. □

**Lemma 3.** *Let $\mathcal{Q}$ be an Oracle that takes an $\mathcal{S} \in SLP[+, \dot{-}, \times, \leftarrow, \rightarrow, Bool]$ and two po-bit positions of $\mathcal{S}$ and determines which position is the more significant. Given access to $\mathcal{Q}$, Oracle $\mathcal{O}$, described in Lemma 2, can be implemented as a polynomial space Turing machine.*

*Proof.* Given an Oracle able to compare between indices, the ability to enumerate over the indices in an arbitrary order allows creation of an ordered enumeration. Essentially, we begin by choosing the smallest value, then continue sequentially by choosing, at each iteration, the smallest value that is still greater than the current value. This value is found by iterating over all index values in an arbitrary order and trying each in turn. □

**Lemma 4.** *Oracle $\mathcal{Q}$, described in Lemma 3, can be implemented as a polynomial space Turing machine.*

*Proof.* Recall that an index position is an affine function of the coefficients of the formal variables introduced, in their standard evaluations. To determine which of two indices is larger, we subtract these, again reaching an affine function of the same form. The coefficients themselves are small, and can be stored directly. Determining whether the subtraction result is negative or not is a problem of the same kind as was solved earlier: subtraction, multiplication and addition need to be calculated over variables; in this case the variables are the coefficients, instead of the original formal variables.

However, there is a distinct difference in working with coefficients, in that they, themselves, are calculable as polynomials over formal variables. The calculation can, therefore, be transformed into addition, multiplication and subtraction, once again over the original formal variables.

Although it may seem as though this conclusion returns us to the original problem, it does not. Consider, among all formal variables, the one defined last. This variable cannot appear in the exponentiation coefficients of any of the new polynomials. Therefore, the new equation is of the same type as the old equation but with at least one formal parameter less. Repeating the process over at most $n$ recursion steps (a polynomial number) allows us to compare any two indices for their sizes. □

*Proof (of Theorem 1).* The equality P-RAM$[+, \leftarrow, Bool]$ = PSPACE was already shown in [12]. Hence, we only need to prove P-RAM$[+, \dot{-}, \times, \leftarrow, \rightarrow, Bool] \subseteq$ PSPACE. This is done, as per Schönhage's method [10], by simulating a polynomial time SLP$[+, \dot{-}, \times, \leftarrow, \rightarrow, Bool]$ on a polynomial space Turing machine.

Lemmas 1–4, jointly, demonstrate that this can be done. □

We remark that Theorem 1 is a striking result, in that right shifting is part of the SLP being simulated, and right shifting is a special case of integer division. Compare this with the power of exact division, described in Theorem 3, which is also a special case of integer division.

### 2.3 Incorporating Arbitrary Numbers

The framework described in Section 2.1 can readily incorporate simulation of arbitrary large number computation. We use it now, to prove Theorem 4.

*Proof (of Theorem 4).* Having proved Theorem 1, what remains to be shown is

$$\text{PTIME-ARAM}[+, \dot{-}, \times, \leftarrow, \rightarrow, \mathit{Bool}] \subseteq \text{PSPACE} . \tag{3}$$

As in the proof of Theorem 1, it is enough to show that an SLP that is able to handle all operations can be simulated in PSPACE.

We begin by noting that because the PTIME-ARAM must work properly for all but a finite range of numbers as its ALN input, it is enough to show one infinite family of numbers that can be simulated properly. We choose $X = 2^\omega$, for any sufficiently large $\omega$. In the simulation, we treat this $X$ as a new formal variable, as was done with outputs of "$a \leftarrow b$" operations.

Lemmas 1–3 continue to hold in this new model. They rely on the ability to compare between two indices, which, in the previous model, was guaranteed by Lemma 4. The technique by which Lemma 4 was previously proved was to show that comparison of two indices is tantamount to evaluating the sign of an affine combination of the exponents associated with a list of formal variables, when using their standard evaluation. This was performed recursively. The recursion was guaranteed to terminate, because at each step the new affine combination must omit at least one formal variable, namely the last one to be defined. Ultimately, the sign to be evaluated is of a scalar, and this can be performed directly.

When adding the new formal variable $X = 2^\omega$, the same recursion continues to hold, but the terminating condition must be changed. Instead of evaluating the sign of a scalar, we must evaluate the sign of a formal expression of the form $a\omega + b$. For a sufficiently large $\omega$ (which we assume $\omega$ to be), the sign is the result of lexicographic evaluation. □

## 3 Models with Division

Our proof of Theorem 3 resembles that of [8] in that it uses Simon's ingenious argument that, for any given $n$, the value $\sum_{i=0}^{2^n-1} i \times 2^{ni}$ can be calculated in O(1)-time by considering geometric series summation techniques. The result is an integer that includes, in windows of length $n$ bits, every possible bit-string of length $n$. The simulating RAM acts by verifying whether any of these bit-strings is a valid tableau for an accepting computation by the simulated TM. This verification is performed using bitwise Boolean operations, in parallel over all options.

Instead of reiterating the entire proof, we give here the most salient differences between the two arguments, these being the places where our argument corrects errors in Simon's original proof. These are as follows.

1. Simon does not show how a TM can simulate an arbitrary RAM in ER-time, making his result a lower-bound only. Indeed, this is, in general, impossible to do. On the other hand, given $\mathrm{EL}_{op}(\mathrm{O}(f(n)))$ tape, a TM can simulate a RAM by storing uncompressed address-value pairs for every non-zero register. The equivalence between the various TMs in Theorem 3 is given by Savitch's Theorem [13], as well as by the well-known relation $\mathrm{TIME}(n) \subseteq \mathrm{SPACE}(n) \subseteq \mathrm{TIME}(\mathrm{EXP}(n))$, so the RAM can be simulated equally well by a time-bounded TM.
2. Simon uses what he calls "oblivious Turing machines" (which are different than those of [14]) in a way that simultaneously limits the TM's tape size and maximum execution time (only the latter condition being considered in the proof), and, moreover, are defined in a way that is non-uniform, in the sense that adding more tape may require a different TM, with potentially more states, a fact not accounted for in the proof. This is corrected by working with non-oblivious, deterministic Turing machines, bounded by a tape of size $s$. Let $c$ be the number of bits required to store the state of the TM's finite control, then

$$\mathit{tape\text{-}contents} + \mathit{state} \times 2^{s+c+\mathit{head\text{-}pos}-1} + 2^{2(s+c-1)+\mathit{head\text{-}pos}} \tag{4}$$

   is a $3(s+c-1)$-bit number encoding the complete instantaneous description of the TM in a way that allows advancing the TM solely by Boolean operations and bit shifting by offsets dependent only on $s$ and $c$. This allows verification of an entire tableau, and, indeed, the entire set of all possible tableaus, simultaneously in $\mathrm{O}(1)$ time, when given $s$, or any number at least as large as $s$, as input. The complexity of the RAM's execution time is due to the $\mathrm{EL}_{op}(\mathrm{O}(f(n)))$ steps required to reach any number that is as large as $s$.
3. Most importantly, Simon underestimates the length needed for the tableau, taking it to be the value of the input. TMs are notorious for using up far more tape than the value of their inputs (see [15]). By contrast, our proof uses the fact that a tape bounded TM has only a finite number of possible instantaneous descriptions, so can only progress a bounded number of steps before either halting or entering an infinite loop. By simulating $2^{3(s+c-1)}$ steps of the TM's execution, we are guaranteed to determine its halting state.

Ultimately, Theorem 3 proves that the power of a RAM[*op*], where *op* is RAM-constructable and includes $\{+, /, \leftarrow, \mathit{Bool}\}$, is limited only by the maximal size of values that it can produce (relating to the maximal tableau size that it can generate and check). Considering this, the proof of Theorem 5 becomes a trivial corollary: instead of generating a number as large as $s$ by $\mathrm{EL}_{op}(\mathrm{O}(f(n)))$ RAM operations, it is possible to assign to $s$ the value of the ALN. Following this single instruction, simulating the TM's entire execution is done as before, in $\mathrm{O}(1)$ time.

We have shown, therefore, that while arbitrary numbers have no effect on computational power without division, with division they provide Turing completeness in $\mathrm{O}(1)$ computational resources.

## References

1. Turing, A.M.: On computable numbers, with an application to the Entscheidungsproblem. Proc. London Math. Soc. 42, 230–265 (1936)
2. Ben-Amram, A.M., Galil, Z.: On the power of the shift instruction. Inf. Comput. 117, 19–36 (1995)
3. Bshouty, N.H., Mansour, Y., Schieber, B., Tiwari, P.: Fast exponentiation using the truncation operation. Comput. Complexity 2(3), 244–255 (1992)
4. Aho, A.V., Hopcroft, J.E., Ullman, J.D.: The Design and Analysis of Computer Algorithms. Addison-Wesley Publishing Co., Reading (1975); Second printing, Addison-Wesley Series in Computer Science and Information Processing
5. Mansour, Y., Schieber, B., Tiwari, P.: Lower bounds for computations with the floor operation. SIAM J. Comput. 20(2), 315–327 (1991)
6. Brand, M.: Does indirect addressing matter? Acta Inform. 49(7-8), 485–491 (2012)
7. Simon, J., Szegedy, M.: On the complexity of RAM with various operation sets. In: Proceedings of the Twenty-Fourth Annual ACM Symposium on Theory of Computing, STOC 1992, pp. 624–631. ACM, New York (1992)
8. Simon, J.: Division in idealized unit cost RAMs. J. Comput. System Sci. 22(3), 421–441 (1981); Special issue dedicated to Michael Machtey
9. Trahan, J.L., Loui, M.C., Ramachandran, V.: Multiplication, division, and shift instructions in parallel random-access machines. Theor. Comp. Sci. 100(1), 1–44 (1992)
10. Schönhage, A.: On the power of random access machines. In: Maurer, H.A. (ed.) ICALP 1979. LNCS, vol. 71, pp. 520–529. Springer, Heidelberg (1979)
11. Hartmanis, J., Simon, J.: On the power of multiplication in random access machines. In: 15th Annual Symposium on Switching and Automata Theory, pp. 13–23. IEEE Comput. Soc., Long Beach (1974)
12. Simon, J.: On feasible numbers (preliminary version). In: Conference Record of the Ninth Annual ACM Symposium on Theory of Computing (Boulder, Colo., 1977), pp. 195–207. Assoc. Comput. Mach., New York (1977)
13. Savitch, W.J.: Relationships between nondeterministic and deterministic tape complexities. J. Comput. System Sci. 4, 177–192 (1970)
14. Pippenger, N., Fischer, M.J.: Relations among complexity measures. J. Assoc. Comput. Mach. 26(2), 361–381 (1979)
15. Radó, T.: On non-computable functions. Bell System Tech. J. 41, 877–884 (1962)