

On the Power of Multiplication  
In Random Access Machines

J. Hartmanis and J. Simon

TR 74-213

September, 1974

Paper presented at the 15th Annual Symposium on Switching and  
Automata Theory, New Orleans, October 14-16, 1974.

Computer Science Department  
Cornell University  
Ithaca, New York 14850



ON THE POWER OF MULTIPLICATION IN RANDOM ACCESS MACHINES<sup>†</sup>Juris Hartmanis  
Janos Simon<sup>††</sup>Computer Science Department  
Cornell University  
Ithaca, N.Y. 14850Abstract

We consider random access machines with a multiplication operation, having the added capability of computing logical operations on bit vectors in parallel. The contents of a register are considered both as an integer and as a vector of bits and both arithmetic and boolean operations may be used on the same register. We prove that, counting one operation as a unit of time and considering the machines as acceptors, deterministic and nondeterministic polynomial time acceptable languages are the same, and are exactly the languages recognizable in polynomial tape by Turing machines. We observe that the same measure on machines without multiplication is polynomially related to Turing machine time--thus the added computational power due to multiplication in random access machines is equivalent to the computational power which polynomially tape-bounded Turing machine computations have over polynomially time-bounded computations. Therefore, in this formulation, it is not harder to multiply than to add if and only if  $PTAPE = PTIME$  for Turing machines. We also discuss other instruction sets for random access machines and their computational power.

1. Introduction

In the theory of computational complexity one tries to classify problems by the amount of resources needed to compute a solution to the problem by some idealized computer. "Popular" computer models are Turing machines (TMs) [10] and random access machines (RAMs) [11], and the amount of resource is usually measured by the number of moves or by the memory used in the computation. One considers both deterministic and nondeterministic models--in addition, the instruction repertory of a RAM may or may not contain indirect addressing, addition, multiplication, bit operations, shifts, etc. Also, it was proposed to charge an amount proportional to the length of the register operated upon for each move of a RAM, instead of a unit cost [3], [1, Ch. 1]. Relationships between these models are central problems in computational complexity and, with the exception of the straightforward ones, largely unknown.

In this paper we consider these machines as acceptors. Moreover, as is customary since [2], we shall pay a lot of attention to polynomial bounds, and will consider two models to

<sup>†</sup> This research was supported in part by grant 70/755 from Fundação de Amparo à Pesquisa do Estado de São Paulo, by Universidade Estadual de Campinas, and by the National Science Foundation Grant GJ-33171x.

<sup>††</sup> On leave of absence from Universidade de Campinas, Campinas, S.P., Brazil.

be essentially equivalent if they are polynomially related.<sup>†</sup>

Within this framework the following is known: deterministic  $T_m$  time (one tape or many tapes) and any reasonable model of a deterministic bounded action machine time are polynomially related [5]. This equivalence class contains also deterministic RAM time (both unit and logarithmic cost) with an instruction set to which we may add indirect addressing, addition and even vector bit operations.

Another equivalence class is constituted by the nondeterministic versions of the above-mentioned machines. Whether the two classes are distinct is the famous  $P = NP$  question [4].

Memory measures form a third class:  $T_m$  tape, number of bits used in a RAM computation (with any reasonable instruction set) are all polynomially related. It is an important result, due to Savitch [9], that the memory (or tape) class also contains the nondeterministic versions of these machines--i.e. nondeterminism gives at most a polynomial reduction in the use of memory. With the exception of this last result, the proofs of all these relationships are straightforward.<sup>††</sup> As we shall see, our techniques yield a new proof of this result.

Very recently Pratt, Stockmeyer and Rabin proved that RAMs with a somewhat unorthodox instruction set, consisting of shift instructions, parallel vector operations and addition (on the index registers) are polynomially equivalent to the third class of memory bounded computations [8]. However, in order to obtain their results, they must partition their registers into two disjoint classes, normal (vector) and shift registers. The only interaction between these is shifting a vector register by the amount contained in a shift register. Arithmetics are limited to additions in the shift register--this model is quite different from the models considered before.

Whether the three classes of machines are polynomially related is not known. Other open problems include the relationships between RAMs with and without multiplication (unit time measure), the relationship between the two time measures for RAMs with multiplication, the relationship between deterministic

<sup>†</sup> Using "translation" techniques, it can be shown that two models are polynomially related if the class of languages accepted in polynomial bound by the first are also accepted in polynomial bound by the second and conversely. See [6] for details.

<sup>††</sup> Some of the proofs may be found in [1].

and nondeterministic RAMs with multiplication and, in general, the amount of power gained by adding features to a RAM's instruction set.

In this paper we obtain the following results: for RAMs with multiplication (and bit operations) nondeterministic and deterministic time models are polynomially related. Note that the same question for RAMs without multiplication is the  $P = NP$  problem. Furthermore, RAMs with multiplication accept in polynomial time exactly the languages accepted by Turing machines in polynomial tape. This implies that the power gained by having multiplication in a RAM--if any--, a problem discussed already in [2], is basically the same as the improvement of  $T_m$  tape over time, another well-known open problem. Also, the two time measures for RAMs with multiplication are polynomially related if and only if memory and time are polynomially related for deterministic TMs. Our results, together with [8], show that a wide range of enlarged instruction set RAMs are polynomially related: we introduce several such instruction sets, and prove their equivalence. The fact that all these machines are equivalent is quite surprising: for example, we will show that the machine introduced in [8] may be simulated in a straightforward manner by allowing multiplication by powers of 2. Since after a polynomial number of steps some of the registers of a RAM with multiplication may contain numbers of exponential length, it is not clear that we may simulate its computation in polynomial time by a RAM which can multiply only by powers of two.

The outline of this paper is the following: Section 1 gives an introduction, Section 2 introduces terminology and notation. In Section 3 we prove half of our main result, namely that we can simulate in polynomial tape a nondeterministic RAM with multiplication operating in polynomial time. This is the hardest proof in the paper: it uses the same ideas as [8] but it is quite a bit more involved. We also show that the result is true even if we add division to the operation set of the RAMs. In Section 4 we sketch a proof of the other half of the result, i.e. that our RAMs can simulate in polynomial time TMs with a polynomial tape bound. We prove this by considering first an instruction set with apparently less power than RAMs with multiplication, show how these may simulate  $T_m$  tape efficiently by using the programming tricks of [8] and show how these machines may be simulated by our other models. The results of these two sections imply that for our RAMs deterministic and nondeterministic time measures and nondeterministic and deterministic  $T_m$  tape measures are all polynomially related. They also show that a wide collection of instruction sets are polynomially related to each other and to  $T_m$  tape. We conclude by stating a few corollaries and making some comments on the meaning of our theorems in Section 5.

## 2. Definitions

A RAM acceptor or RAM with instruction set  $O$  is a set of registers  $R_0, R_1, \dots$  each capable of storing a non-negative integer in

binary representation, together with a finite program of (possibly labeled)  $O$ -instructions. If no two labels are the same, we say that the program is deterministic, otherwise it is nondeterministic. We call a RAM model deterministic if we consider only deterministic programs from the instruction set.

Our first instruction set consists of the following:

$O_1$

$R_i + R_j$	(= $k$ )	(assignment)
$R_i + \langle R_j \rangle$		(indirect addressing)
$R_i + R_j + R_k$		(sum)
$R_i + R_j - R_k$		(proper subtraction)
$R_i + R_j$ <u>bool</u> $R_k$		(boolean operations)
if $R_i$ <u>comp</u> $R_j$ label 1		(conditional jump)
else label 2		
<u>accept</u>		
<u>reject</u>		

comp may be any of  $<, \leq, =, \geq, >, \neq$ . For boolean operations we consider the integers as bit strings and do the operations componentwise. Leading 0s are dropped at the end of operations: for example,  $11$  nand  $10 = 1$ . bool may be any binary boolean operation (e.g.  $\wedge, \vee, \text{eor}, \text{nand}, \text{or}$ , etc.). accept and reject have obvious meanings. An operand of  $=k$  is a literal and the constant  $k$  itself should be used.

The computation of a RAM starts by putting the input in register  $R_0$ , setting all registers to 0 and executing the first instruction of the RAM's program. Instructions are executed in sequence until a conditional jump is encountered, after which one of the instructions with label "label 1" is executed if the condition is satisfied and one of the instructions with label "label 2" is executed otherwise. Execution stops when an accept or reject instruction is met. A string  $x \in \{0,1\}^*$  is accepted by the RAM if there is a finite computation ending with the execution of an accept instruction. The complexity measures defined for RAMs are:

(unit) time measure: the complexity of an accepting computation is the number of instructions executed in the accepting sequence. The complexity of the RAM on input  $x$  is the minimal complexity of accepting computations.

logarithmic, or length time measure: the complexity of an accepting computation is the sum of the lengths of the operands of the instructions executed in the accepting sequence. When there are two operands, we take the length of the longer; when an operand has length 0 we use 1 in the sum. The complexity of the RAM on input  $x$  is the minimal complexity among accepting computations.

memory measure: the maximum number of bits used at any time in the computation. (The number of bits used at a given time is the sum of the number of significant bits of

all registers in use at that time.)

Unless otherwise stated, time measure will mean unit time measure. We shall call RAMs with instruction set  $O_1$   $RAM_1$ s or simply RAMs. For a discussion of RAM complexity measures, see [1] or [3].

These definitions are standard, with the exception of the boolean operators. We argue however that the reason they were left out of earlier RAM definitions (where + was an operator) was mainly because RAMs were mostly used to represent von Neumann computers working on numerical problems. All real computers have such capabilities, so if RAMs are to be more or less realistic models of them, they should have boolean operations. Anyhow, if we define RAMs with an instruction set consisting of  $O_1$  minus the boolean operators, call it  $O_0$ , then RAMs with instruction set  $O_0$ ,  $RAM_0$ s, are polynomially related to  $RAM_1$ s in all measures.

This can be proved easily for the unit time measure by noting that one can compute a boolean function of two bit arguments on a  $RAM_0$  and a boolean function of a bit vector in time proportional to the length of the operands. Since the latter may increase at most by one per operation, the result follows.

We will consider other instruction sets:

$O_2$  is  $O_1$  plus the instruction

$$R_i \leftarrow R_j \circ R_k \quad (\text{concatenation})$$

which leaves in  $R_i$  the contents of  $R_j$  followed by the contents of  $R_k$ . Again, the operands may be literals. We shall call RAMs with instruction set  $O_2$  CRAMs (C for concatenation).

$O_3$  is  $O_1$  plus the instruction

$$R_i \leftarrow R_j \cdot R_k \quad (\text{product})$$

which computes the product of the two operands (which may be literals) and stores it in  $R_i$ . RAMs with instruction set  $O_3$  will be called MRAMs (M for multiplication).

$O_4$  is  $O_3$  plus the instruction

$$R_i \leftarrow R_j \div R_k \quad (\text{integer division})$$

which leaves in register  $R_i$  the result of dividing  $R_j$  by  $R_k$  and taking the integer part of the result. If  $R_k$  contains 0 the machine jams and rejects. These RAMs will be called PRAMs (P for powerful).

Finally, we describe VRAMs, defined in [8]. As we mentioned before, this model is quite different from the previous ones. There are two different kinds of registers: shift registers and general (vector) registers. The only interaction between the two is by means of the shift instructions

$$V_i \leftarrow I_k \quad (\text{shift right})$$

$$V_i \leftarrow I_k \quad (\text{shift left})$$

which shift the contents of general register  $V_i$  to the right or left by the amount contained in shift register  $I_k$ . For shift registers we have the instructions of assignment, sum, proper subtraction and division by 2; for general registers we have only boolean operations. In addition, we have conditional jumps using the result of a comparison between two general registers or between two index registers to decide which label to jump to. Literals and indirect addressing may be used in all operations.

Our  $T_m$  model is the off-line  $T_m$  of [10]: a finite control, a read-only input tape and a read-write work tape. Time measure is the number of moves and tape measure the longest work tape used in the accepting computation (for nondeterministic models we take the minimum among accepting computations).

Finally, we define the class PTIME -  $\langle \text{machine} \rangle$ , where  $\langle \text{machine} \rangle$  may be  $T_m$ ,  $RAM$ ,  $RAM_0$ , CRAM, MRAM, PRAM or VRAM as the class of languages for which there is a deterministic machine which accepts the language within a polynomial number of steps. The class  $PTAPE - \langle \text{machine} \rangle$  will designate the class of languages accepted in polynomial storage. We shall use the prefix "N" to designate non-deterministic models. We also use "P" or "PTIME" for  $PTIME - T_m$ , and "PTAPE" for  $PTAPE - T_m$ .

As we mentioned before, the following is true:

Lemma: 1)  $P = PTIME - RAM = PTIME - RAM_0$

Moreover, if we define length - PTIME to denote the class PTIME in the length measure,

$$\begin{aligned} P &= \text{length} - PTIME - RAM \\ &= \text{length} - PTIME - CRAM \\ &= \text{length} - PTIME - MRAM \\ &= \text{length} - PTIME - VRAM \end{aligned}$$

$$\begin{aligned} 2) NP &= NPTIME - RAM \\ &= NPTIME - RAM_0 \\ &= \text{length} - NPTIME - RAM \\ &= \text{length} - NPTIME - CRAM \\ &= \text{length} - NPTIME - MRAM \\ &= \text{length} - NPTIME - VRAM \end{aligned}$$

$$3) PTAPE = NPTAPE = PTAPE - RAM = NPTAPE - RAM$$

for all RAM models.

We shall prove in the next section that

Theorem 1:  $PTAPE \supseteq NPTIME - MRAM$

In Section 4 we show that

Theorem 2:  $PTIME - CRAM \supseteq PTAPE$ .

In the same section we also show that

$$\begin{aligned} NPTIME - MRAM &\supseteq PTIME - MRAM \\ PTIME - MRAM &\supseteq PTIME - VRAM \\ PTIME - VRAM &\supseteq PTIME - CRAM \end{aligned}$$

All these containments are straightforward. The set of containments implies our main results

NPTIME - MRAM = PTIME - MRAM  
and  
PTIME - MRAM = PTAPE.

It also means that all of the following coincide:

PTAPE,  
PTIME - CRAM, NPTIME - CRAM  
PTIME - MRAM, NPTIME - MRAM  
PTIME - VRAM, NPTIME - VRAM  
PTIME - PRAM, NPTIME - PRAM.

The last line follows from the proof, at the end of Section 3, that PRAMs may be simulated in polynomial time by MRAMs. Note that since Theorem 1 actually says (deterministic)  $PTAPE \supseteq NPTIME - MRAM$ , and Theorem 2 that  $PTIME - CRAM \supseteq NPTIME - VRAM$ , we get a new proof of Savitch's result [9] that  $PTAPE = NPTAPE$ .

It should be pointed out again, that the main result  $PTIME - MRAM = PTAPE$

in conjunction with the observation that  $PTIME - RAM_0 = PTIME$

characterizes the additional power gained by (one-step) multiplication in RAMs as the difference, if any, between PTIME and PTAPE for Turing machine computations.

### 3. $PTAPE \supseteq NPTIME - MRAM$

In this section we prove our main theorem, the simulation of MRAMs working in polynomial time by Turing machines using polynomial tape. We shall not attempt a very efficient simulation, but we try to make the construction as clear as possible.

Suppose the MRAM  $M$  operates in time  $n^k$ , where  $n$  is the length of the input. Our  $T_m$  simulator  $T$  will write out in one of its tapes a guess for the sequence of operations executed by  $M$  in its accepting computation and check that the sequence is correct. The sequence may be written down deterministically, by enumerating all such sequences of length  $n^k$  in alphabetical order. Since the number of instructions of  $M$ 's program is a constant, the sequence will be of length  $cn^k$  for some constant  $c$ . To verify that such a sequence is indeed an accepting computation of  $M$  we need to check that one step follows from the next when  $M$ 's program is executed -- which is only a problem in the case of conditional instructions, when we must find out the contents of a register. We shall define a function  $FIND(r, b, t)$  which will return the value of the  $b$ -th bit of register  $r$  at time  $t$ . Our theorem will be proved if this function is computable in polynomial tape -- the subject of the remainder of this section. Note that since we are testing for an accepting sequence, it does not matter whether we are simulating deterministic or nondeterministic machines.

First, let us prove that the arguments of  $FIND$  may be written down in polynomial tape. Note that in  $t$  operations the biggest possible number that may be generated is  $a^{2^t}$ , produced by successive multiplications:  $a, a^2,$

$a^2 \cdot a^2 = a^4, a^4 \cdot a^4 = a^8, \dots, a^{2^t}$  where  $a$  is the maximum of  $x$  and the biggest literal in  $M$ 's program. To address a bit of it, we need to count up to its length, that is, up to  $\log_2(a^{2^t}) = 2^t \log_2 a$ , which may be done in space  $\log_2(a^{2^t \log_2 a})$ . In particular, for  $t = n^k$ , space  $n^{k+1}$  will suffice, so that  $b$  may be written down in polynomial tape.

Clearly,  $t$  may also be written down in polynomial tape.

There is a small difficulty with  $r$ : since we allow indirect addressing, although in time  $n^k$  at most  $n^k$  registers are accessed, the address of a register may be as high as  $2^{2^t}$ , which has length  $2^t$  and cannot therefore be written in polynomial tape. However, at the cost of at most a square factor in time, we may restrict an MRAM operating in time  $t$  to use only its first  $t$  registers:

Let  $M'$  be an arbitrary MRAM.  $M''$  will mimic  $M'$  but use only its first  $2t$  registers.  $M''$  uses its registers in pairs: the first component of the register pair holds an address, the address of a register of  $M'$ ; the second component has the actual contents of that register. When  $M''$  has to simulate a move of  $M'$  which accesses the register  $s$ ,  $M''$  first determines whether a first component holding  $s$  exists among the first  $t$  register pairs of  $M''$ . If so,  $M''$  accesses the second component of that pair. Otherwise,  $M''$  creates a new pair in the first two available locations by storing  $s$  in the first component (register) and using the second for the  $s$ -th register of  $M'$ . Clearly the simulation of a move of  $M'$  takes at most  $ct$  steps for some constant  $c$ , so that  $M''$  operates in time  $ct^2$ . It uses only its first  $2t$  registers.

We shall suppose that  $M$  uses only its first  $n^k$  registers. We have shown that in that case all arguments of  $FIND$  may be written down in polynomial space.

Now let us describe  $FIND$  and prove that it operates in polynomial tape.

Informally,  $FIND$  works as follows:  $FIND(r, b, 0)$  is easily computed given the input. We shall argue inductively.  $FIND(r, b, t)$  will be computed from previous values of  $FIND$  -- clearly the only interesting case is when  $r$  was altered in the previous move. For example, if the move at  $t-1$  was  $r + pVs$ , then  $FIND(r, b, t) = FIND(p, b, t-1) \vee FIND(s, b, t-1)$ . This recursion in time does not cause any problems, because we may first compute  $FIND(p, b, t-1)$  and then reuse the tape for a call of  $FIND(s, b, t-1)$ , so that if  $l_{t-1}$  is the amount of tape needed to compute  $FIND$ s for times up to  $t-1$ , we have the recurrence

$$l_t = l_{t-1} + c \quad (l_0 = cn^{k+1})$$

which has the solution  $l_t = c'n^{k+1}$ .

In the case of shift machines, studied

in [8], this is the only recursion necessary. However, with our machines, in the case of multiplication of two  $l$ -digit numbers, we may have to compute up to  $l$  factors and get the carry from the previous column in order to obtain the desired bit. Since  $l$  may be

$2^n$ , we must be able to take advantage of the regularity of operations in order to be able to compute within polynomial tape. Also, the carry from the previous column may be quite big: in the worst case, when we multiply

$(1)^l$  by  $(1)^l$  the carry may be  $l$ . This is still manageable, since in time  $n^k$ ,  $l \leq 2n^k$ ; an accumulator of length  $n^k$  will suffice. We also need to generate up to  $l$  pairs of bits, multiply them in pairs and add them up. This may be done as follows: we store the addresses of the two bits being computed, compute each of the two bits of the product separately multiply the two results and update the addresses to get the addresses of the two bits of the next product. The product is added to an accumulator and the process is repeated until all product terms have been computed. Then we need the carry from the previous column.

We cannot compute this carry by a recursive call of FIND, because since the length of the register may be exponential, keeping track of the recursion would take exponential tape. Instead, we compute the carries explicitly from the bottom up -- i.e., we first compute the carry at the rightmost column (finding the bits by recursive calls of FIND on pairs and multiplying them), and then, with that carry and FIND, we compute the carry from the second rightmost column, and so on. The space needed is only for keeping track of which column we are at, one recursive call of FIND, one accumulator and one previous carry holder. Each of these may be written down in space  $n^{k+1}$ , so that we have the recursion

$$l_t = l_{t-1} + cn^{k+1}, \text{ with } l_0 = n^{k+1}$$

which implies  $l_t \leq cn^{2k+1}$ ,

and the simulation of multiplication may be carried out in polynomial space.

The argument for + is similar but much easier, since only 2 bits and a carry of at most 1 are involved.

A bastard PL/I (PL/B?) of FIND follows:

```
FIND: PROCEDURE (r,b,t), returns (digit)
/* We omit the trivial code for t=0 */
/* We suppose FIND has access to global
variables that specify M's action at
all times */
if instruction at time t-1 not of the
form q + p op s then return (FIND
(r,b,t-1)) fi
if r ≠ q then return (FIND(r,b,t-1))
/* register was not modified at time
t-1 */
else
if op = boolean operation
then /* compute relevant bits from
operands */
BIT 1 = FIND(p,b,t-1)
BIT 2 = FIND(s,b,t-1)
return (BIT 1 op BIT 2)
```

```
else
if op = .
then /* loop through columns
until current one is
reached */
COLUMN = 0
CARRY = 0
while COLUMN < b do
FIRSTPTR = 0 /* addresses
of
bits to
be multi-
plied*/
SECONDPTR = COLUMN
ACUM = 0
while SECONDPTR > 0 do
/* add up products
in ACUM */
BIT 1 = FIND (p,FIRST-
PTR, t-1)
BIT 2 = FIND (s,SECOND-
PTR, t-1)
ACUM = ACUM+BIT 1·BIT 2
FIRSTPTR = FIRSTPTR+1
SECONDPTR = SECONDPTR
- 1
end
ACUM = ACUM + CARRY
/* get total sum in
column */
CARRY = if ACUM>0 then
(ACUM - 1)/2
else 0 /*shift
right by 1 */ fi
end
return (ACUM mod 2)
else /* op = + */
/* compute carries from right
to left, as for . */
COLUMN = 0
CARRY = 0
while COLUMN < b do
ACUM = 0
BIT 1 = FIND (p,COLUMN,t-1)
BIT 2 = FIND (s,COLUMN,t-1)
ACUM = BIT 1+BIT 2 + CARRY
CARRY = if ACUM > 0 then
(ACUM - 1)/2 else
0 fi
end
return (ACUM mod 2)
fi
end: FIND.
```

Let us analyze the tape requirements of FIND.

All the inputs are representable in tape  $n^{k+1}$ . Moreover, no loop control variable exceeds an input variable; the same is true of FIRSTPTR and SECONDPTR. Also, we saw that the greatest possible carry is of order  $2n^k$ , and therefore representable in tape  $n^k$ . Therefore all variables are representable in space  $cn^{k+1}$  in any activation of FIND. The only possible problem arises with the recursion: however note that we have only one active call at a time in every activation of FIND and it has its  $t$  parameter smaller by one than the  $t$  of its calling routine. Thus, at most  $n^k$  activations of FIND may be present at any given time, and since each of them occupies at most  $cn^{k+1}$  squares of tape, the whole procedure

works in space  $cn^{2k+1}$ .

This ends the proof of our theorem.

The features of FIND that carry the proof through are:

1) the possibility of computing the relevant digits of results of previous computations one at a time; even though there is an exponential number of them, the rule for their formation is easy.

2) the fact that the the carry may be computed explicitly, in an orderly fashion from right to left. In this way, the only information needed from one column to the next is the carry from the previous column which, luckily, is just small enough to be representable in polynomial space.

For the benefit of the reader who got lost in the details: we have proven

**Theorem 1:** Polynomial time bounded non-deterministic MRAM-recognizable languages are recognizable in polynomial time by Turing machines (deterministically).

In the next section we show the converse.

In the remainder of this section we extend the simulation to PRAMs. First note that a straightforward extension of the technique used to prove Theorem 1 fails: to compute the carries in a bit-by-bit simulation of the division algorithm we may need exponential space, a fact that the reader may want to verify by himself. However, in [1, Ch. 8] an algorithm is presented which, given an  $n$ -bit integer  $p$ , computes  $2^{2n-1}/p$  in  $O(\log n)$  operations. This number is basically the reciprocal of  $p$ : to find  $[a/p]$  we find  $1/p$ , multiply by  $a$  and shift by an appropriate amount. A shift corresponds to a division by 2, for which, unlike general division, our techniques of simulation by TMs do work. The computation of the reciprocal is done by a recursive technique: it is easy to get the first (most significant) digit of  $b = 1/p$ . At stage  $i$ , we have an approximation  $b_i$  to  $b$  satisfying

$$b = b_i + (1/p)(1 - b_i p).$$

Using  $b_i$  as an approximation for  $1/p$ , we obtain the recursive formula

$$b_{i+1} = b_i + b_i(1 - b_i p).$$

Note that the method converges quadratically and may be programmed to yield  $1/p$  to  $2k$  bits from an approximation to  $k$  bits in a constant number of operations (Algorithm 8.1 in [1]).

To obtain the result  $[a/p]$ , we need to compute  $1/p$  to an accuracy of  $\log_2 a + 1$  digits. Since, as we saw, in  $t$  operations  $a$  is of order  $2^{2^t}$ , we need to get  $2^{2^t} + 1$  bits of  $1/p$ , which may be done in  $O(t)$  operations. Thus an MRAM acceptor may simulate a PRAM acceptor with a loss of efficiency of at most a square factor.

#### 4. $\text{PTAPE} \subseteq \text{PTIME} - \text{MRAM}$

In this section we outline a proof that polynomial tape bounded  $T_m$  acceptable

languages are recognizable in polynomial time by MRAMs, i.e.

**Theorem 2:**  $\text{NPTAPE} \subseteq \text{PTIME-CRAM} \subseteq \text{PTIME} - \text{MRAM}$ .

The proof is based on a collection of programming tricks (or fast algorithms) and we only outline the techniques, hoping that the interested reader will be capable of filling in the details. Complete proofs (at least almost complete) for the VRAM case may be found in [8]. We shall use CRAMs in our constructions, augmented, for simplicity, by the instruction SUBSTR. SUBSTR(A,B) produces A with its initial substring of length length(B) deleted (e.g. SUBSTR(10011,11) = 011). Later we shall show how to do without this operator.

The essential idea of the proof is the following:

for any given  $T_m$ ,  $T$ , operating in polynomial time on input  $x$ , a CRAM can first generate all possible configurations of this  $T_m$  computation (a configuration of  $T$  on input  $x$  consists of the state of  $T$ , the contents of the work-tape and the positions of  $T$ 's heads). From this set of all possible configurations, the CRAM can obtain the matrix of the relation "follow in one move" -- i.e. if  $A$  is the matrix of the relation then  $a_{ij} = 1$  iff  $T$  passes from the  $i$ -th to the  $j$ -th configuration in one move. Clearly,  $x$  is accepted by  $T$  iff  $a_{be}^* = 1$  where  $A^*$  is the transitive closure of  $A$  and  $b$  and  $e$  are initial and accepting final configurations respectively. To make matters simple we shall suppose without loss of generality that  $T$  has only one accepting configuration  $e$ . We shall see that parallel bit operations, together with operations that expand rapidly the length of a register, enable us to do each of these steps in polynomial time on a CRAM.

First we indicate how to compute efficiently the transitive closure of a matrix  $A$ . We suppose that initially the whole matrix is in a single register. Remember that  $A^* = I \vee A \vee A^2 \vee A^3 \vee \dots \vee A^n \vee \dots$ , where  $A$  is  $n$  by  $n$  and  $A^i$  is the  $i$ -th power of  $A$  in the "and-or" multiplication (i.e. if  $C = A \cdot B$ ,  $c_{ij} = \bigvee_{k=1}^n a_{ik} \wedge b_{kj}$ ). Moreover, we may compute the products  $(I \vee A), (I \vee A)^2, (I \vee A)^2 \cdot (I \vee A)^2 = (I \vee A)^4, \dots$  where the exponent of  $(I \vee A)$  is a power of 2. Since there are only  $\log n$  of these  $((I \vee A)^{n+k} = (I \vee A)^n$ ) transitive closure of  $n$  by  $n$  matrices can be done in time  $\log n$  times the time for multiplication. Throughout this section, "multiplication" will mean " $\wedge$ " and "multiplication of matrices", "and-or" multiplication. Also, for simplicity, we assume  $n$  to be a power of 2.

To multiply two matrices efficiently, we observe that if we have several copies of the matrix stored in the same register in a convenient way, we can obtain all products in a single " $\wedge$ " operation: all we need is that for all  $i, j$ , and  $k$ ,  $a_{ik}$  be in the same bit position as  $b_{kj}$ . For example, if we have



(row 0 of A)<sup>n</sup> (row 1 of A)<sup>n</sup> ... (row n-1 of A)<sup>n</sup> =  
 $(a_{0,0}a_{0,1}\dots a_{0,n-1})^n (a_{1,0}a_{1,1}\dots a_{1,n-1})^n \dots$   
 $(a_{n-1,0}a_{n-1,1}\dots a_{n-1,n-1})^n$  in one register  
 (where (row i)<sup>n</sup> means n-fold concatenation) and  
 [(column 0 of B) (column 1 of B) ...  
 (column n-1 of B)]<sup>n</sup> =  $[(b_{0,0}b_{1,0}\dots b_{n-1,0})$   
 $(b_{0,1}b_{1,1}\dots b_{n-1,1}) \dots (b_{0,n-1}\dots b_{n-1,n-1})]$ <sup>n</sup>  
 In the other, the "Λ" of the two registers  
 yields all terms  $a_{ik} \wedge b_{kj}$ . Supposing we are  
 able to produce these forms of the matrices  
 easily, all we have to do is collect terms and  
 add (V) them up. To collect terms, if we are  
 able to take advantage of the parallel opera-  
 tions at their fullest, we should not have to  
 do more than log n operations, since each  $c_{ij}$   
 is the sum of n products. Note that in our  
 case  $c_{0,0}$  is the sum of the first n bits,  $c_{0,1}$   
 of the next n, and in general  $c_{ij}$  is the sum  
 of bits  $i \cdot n + j \cdot n$  to  $i \cdot n + (j+1)n$ .

We use the following idea: to add up a  
 row vector of bits, take the second half of  
 the row, add it in parallel to the first half  
 and call the procedure recursively for the new  
 first half. The reader is encouraged to write  
 a routine, using the mask  $M' = 0^{n/2}1^{n/2}$  to se-  
 lect the second half (n is the length of the  
 vector) and prefixing strings of 0s to regis-  
 ters to get proper alignment. It is possible  
 to design the algorithm in such a way that  
 this procedure may be done in parallel for  
 several vectors, stored concatenated to each  
 other in a single register. In particular if  
 one starts with  $n^2$  copies of the mask  $M'$ , then  
 the following procedure obtains all terms of  
 the matrix product  $C = A \cdot B$  from all the prod-  
 ucts  $a_{ik} \wedge b_{kj}$ :

```

ADDUP: PROC
    M = (0n/2. 1n/2)n2
    K = n/2
    while K > 1 do
        B = AAM
        A = ((0K.A)VB)AM
        K = K/2
        M = (0K.M)AM
    end
end:ADDUP
    
```

ADDUP uses 0<sup>K</sup> and K/2 as primitive operations,  
 but K/2 = SUBSTR(K,1) and 0<sup>K</sup> may be obtained  
 by successive concatenations of a string with  
 itself: after p steps we get a string of 0s  
 of length 2<sup>p</sup>.

In order to perform matrix multiplica-  
 tions, we have to show that we can efficiently  
 expand a matrix from some standard input form  
 (say stored by rows) into the forms used in  
 forming the product. The idea is again the  
 same: use masks and concatenations to get  
 lots of elements in the places where we need  
 them, in parallel. For example, to get

(row 0)<sup>n</sup>(row 1)<sup>n</sup>... (row n-1)<sup>n</sup>  
 from (row 0)(row 1) ... (row n-1)  
 we first take the second half away, so that

we may put the middle row in its final place,  
 using the mask  $M' = 0^{n^2/2}1^{n^2/2}$ ;

$B = M' \wedge A = 0 \dots 0 a_{n/2,0} a_{n/2,1} \dots a_{n-1,n-1}$   
 We have to slide B under A in such a way that  
 $a_{n/2,0}$  occupies the  $n^3/2$ -th bit position.

```

To put B in its place, all we need is:
z = 0n2/2 †
SHIFT = 0n2/2
K = n
while K > 1 do
    SHIFT = SHIFT.SHIFT
    K = K/2
end /*SHIFT is now 0n3/2 */
    
```

SHIFT = SUBSTR(SHIFT,Z) /\* now SHIFT is  
 $0^{n^3/2 - n^2/2}$  -- exactly the  
 amount we need to move B by \*/

We get the desired result by setting  $A =$   
 $(A \wedge M')VB = a_{0,0}a_{0,1}\dots a_{n/2-1,n-1} 0 \dots a_{n/2,0}$   
 $a_{n/2,1}\dots a_{n-1,n-1}$  position  
 $n^3/2$

This is the first step in our method, but  
 it is reasonably clear how to proceed: the  
 next mask should be

$(0^{n^2/4} 1^{n^2/4} 0^{n^2/4} 1^{n^2/4})^n$ ,  
 that we may obtain by:  
 $N = (0^{n^2/4}.M') \text{ eor } M'$   
 $M' = \text{SUBSTR}(N, 0^{n^2/4})$

and the second halves set  $B = AAM'$  will have  
 to be shifted by  $n^2/4 (n-1)$  (remember, the  
 previous iteration shifted B by  $n^2/2 (n-1)$ ).

This will put rows  $n/4$  and  $3n/4$  in their pla-  
 ces. The reader should have no difficulty  
 writing down an efficient program which pro-  
 duces:

$A = (\text{row } 0)0^{n^2-n}(\text{row } 1)0^{n^2-n}\dots(\text{row } n-1)$   
 (the program outlined is not optimal and will  
 run in time  $O((\log n)^2)$ ).

Now it is fairly easy to get the matrix in  
 the desired form in  $O(\log n)$  duplications.

Basically the same trick works to obtain  
 the column form from the stored-by-row form:  
 first we produce, as before, the form

(row 0)<sup>n</sup>0<sup>n-1</sup>(row 1)<sup>n</sup>0<sup>n-1</sup>... (row n-1)  
 from which we get (using the same technique)  
 $a_{0,0}0^{n-1}a_{0,1}0^{n-1}a_{0,2}0^{n-1}\dots a_{0,n-1}0^{n-1}a_{1,0}0^{n-1}$   
 $\dots a_{n-1,n-1}$   
 (i.e. position of  $a_{ij} = n^2i + nj$ ), from which,

† Strictly speaking, this is illegal.  $0^k = 0$   
 in our RAMs. However we may use  $1^k$  and, in  
 concatenations do  $(1^k.A) \text{ eor } 1^k = 0^k.A$ , so that  
 we will use  $0^k$  as an abbreviation.

by using the mask  $0^{n^3/2} 1^{n^3/2}$  and again the same tricks, one obtains A in column order. Concatenation of this with itself  $\log n$  times gives us the form needed for matrix multiplication in  $O(\log n)$  operations.

We would like to emphasize that the routines presented above are not the most efficient or most economical in terms of storage. We just wanted to give a hint of the basic technique and hope that the interested reader will be able to derive the complete programs himself, by using the tricks shown. In any case, we consider that we have outlined a proof that transitive closure may be computed in  $O((\log n)^2)$  moves.

We still have to convince the reader that given a polynomial tape bounded  $T_m$  with input  $x$ , we can obtain the matrix of the "follow in one move" relation easily. We shall do this in an even sketchier way than our exposition of the method for computing transitive closure.

If a  $T_m$  operates on an input of length  $n$  in tape  $n^k$ , there are at most  $O(2^n)$  different configurations. Let us take a convenient encoding of these in the alphabet  $\{0,1\}$  and interpret the encodings as integers. By convenient encoding we mean one that is linear in the length of the tape used by the machine, where the positions of the heads and the state may be easily found, and which may be easily updated. Then, if we generate all the integers in the range  $0 - (2^{cn} - 1)$  (where  $c$  depends only on the encoding) we shall have produced encodings of all configurations, together with numbers that are not encodings of any configuration. The reader might amuse himself by writing a CRAM program that produces all integers between 0 and  $m = 2^p - 1$  in time  $p$ . (Hint: for a straightforward program get  $(m = 2^p) \ 1^{m/2} 0^{m/2}$ , then  $(10^{p-1})^{m/2}$ , then  $01^{m/4} 0^{m/4} 1^{m/4} 0^{m/4}$  and  $0(10^{p-1})^{m/4} 0^{m/4} (10^{p-1})^{m/4}$ ,  $\dots$  V them together, etc.)

Now, it is well known that in the operation of the  $T_m$  the character under the read-write head, the two symbols in the squares immediately to the right and left of it, the state of the finite control and the position of the input head determine any next configuration. Then we test whether the configuration  $c_j$  follows from  $c_i$  as follows: suppose  $c_i$  and  $c_j$  are stored in registers R and S. We first build a pattern which picks up head positions (i.e., once we build the pattern, we obtain from R and S, in a constant number of moves, bit vectors which have a 1 at the position scanned by the head and 0s everywhere else -- moreover the sequence of moves is independent from the contents of R and S. For example, suppose that the head position is indicated by the pattern 11011 appearing beginning at some position  $p = 0 \pmod{5}$  in the encoding of the configuration, which we suppose of length  $\ell' = 5\ell$ . Then  $M_h = (00100)^\ell$  is a mask with the property that  $T = M_h \text{ eor } R$  will have 11111 starting at a position  $p \equiv 0 \pmod{5}$  iff R had 11011 there. Using a procedure similar to ADDUP, we get a vector which is 1 only

at such positions and 0 everywhere else). Now, again in a manner that does not depend on where the head is in  $c_i$ , we may, in another constant number of moves, obtain the three squares of R that matter for the determination of the next configuration, as well as the state of the finite control. We save this information and zero the corresponding bits in the encodings, both in R and S. All of this can be done in a constant number of moves, which are independent of the contents of R and S.

To verify that the transition was a permissible move of the  $T_m$  we have to check that the non-blank portions of R and S are identical and that the blanked-out bits satisfy a move rule. The latter is verified by table look-up, where the size of the table depends only on the  $T_m$  but not on the input, while the former is checked by first taking  $R \text{ eor } S$  (R and S have now 0s where a move might change R) and using a version of ADDUP to verify that the result consists only of 0s. This will take only  $O(\log n)$  moves.

Thus, we know how to detect the fact that  $c_j$  follows from  $c_i$  in  $O(\log \ell)$  CRAM moves, where  $\ell$  is the length of the configuration and the moves do not depend on the contents of  $c_i$  or  $c_j$ . This is important, because it shows that if we have  $c_{i_0} c_{i_1} \dots c_{i_k}$  in R,  $c_{j_0} \dots c_{j_k}$  in S, we may, in  $O(\log \ell)$  moves, test simultaneously whether  $c_{j_k}$  follows from  $c_{i_k}$ . Now, the way to generate the transition matrix in time  $O(\log n)$  where  $n$  is the length of the input is easy enough to guess: first we generate all integers in the range

$0 - (2^{n^k} - 1)$ , call these configurations  $c_i$ . Then, as in the matrix product routine, we form  $(c_0)^m (c_1)^m \dots (c_{m-1})^m$

where  $m = 2^{n^k}$  and  $(c_i)^m$  means  $m$ -fold concatenation, and  $(c_0 c_1 \dots c_{m-1})^m$  in  $O(\log m) = O(n^k)$  operations, and in  $O(n^k)$  operations determine simultaneously for all  $i$  and  $j$  whether  $c_j$  follows from  $c_i$  (i.e. obtain a vector of bits which is 1 iff  $c_j$  follows from  $c_i$ ). This completes the description of our simulation algorithm: putting everything together we have a procedure which runs in polynomial time, since

the matrix may be computed in  $O((\log 2^{n^k})^2)$  moves and its transitive closure in

$O((\log 2^{n^k})^2) = O(n^{2k})$  moves.

This completes the outline of the proof for the special CRAM used.

Finally, some comments about the instruction sets necessary to do this simulation. In our programs we used, besides parallel bit operations, the following: concatenation ( $\cdot$ ), SUBSTR, and loop control operations (comparisons and divisions by 2). We first show, as we have promised, how to eliminate SUBSTR. The basic idea is simple, and we used it implicitly in ADDUP: the SUBSTR operation is

used to drop off an initial substring of a string to obtain alignment -- but the same effect can be obtained by concatenating a string of 0s of the same length to the other string. This has the disadvantage that now we have a certain amount of useless garbage preceding certain variables, but that can be taken care of by the following:

first, it is easy to see that we may always assume that the initial segment is a string of 0s, since for any prefix P,  $(P.A) \text{eor } P = 0^{\text{length}(P)}.A$ ;

second, we maintain, for each variable, an associated "garbage indicator" -- another register which contains a string of 1s of length equal to the useless initial segment. Whenever a variable with a nonempty garbage indicator is used in conjunction with others, if the operation is a boolean one we prefix the other operand with the garbage indicator transformed into 0s. If we want to form  $C=A.B$

but we have only  $A' = 0^{n_1}.A$ ,  $B' = 0^{n_2}.B$ ,

$G_A = 1^{n_1}$ ,  $G_B = 1^{n_2}$ , we form  $C = A'.B'$

$(= 0^{n_1}.A.0^{n_2}.B)$

$C = C \text{eor } A' (= 0^{n_1+n_2+\text{length}(A)}$

$C = C \vee ((G_B.A) \text{eor } G_B) (= 0^{n_1+n_2}.A.B)$

$G_C = G_A.G_B$ .

In both cases we have only a constant amount of overhead per operation. Thus SUBSTR is not necessary.

As for the loop control, it is again easy to see that division by 2 is not necessary, since it is SUBSTR(,1). Thus the main theorem of this section may be written:

CRAMS without arithmetic instructions may simulate PTAPE in polynomial time.

We now sketch proofs of how our more powerful RAM models may simulate CRAMS.

#### 1) VRAMS

Clearly  $A.B$  is the same as  $A \vee (B + \text{length}(A))$ . All we have to show is that the necessary lengths are attainable in polynomial time for polynomial time bounded CRAM computations.

Initially we store the lengths of all constants used in the CRAM program in the VRAM's program. The length of the input may be obtained in linear time. The longest string obtainable in  $t$  moves from a set  $S$  of strings, by a CRAM, is  $2^t \text{length}(S_0)$ , where  $S_0$  is the longest string in  $S$ . It is easy to devise a binary search type VRAM algorithm that will find the length of a string in time  $O((\log \text{length}(x))^2)$ .

Thus, if a CRAM operates in time  $n^k$ , we may simulate each of its steps in at most  $O(n^{2k})$  VRAM steps and, therefore, the whole computation in time  $O(n^{3k})$ . Again, the simulation technique is not optimal, but, we hope, transparent.

#### 2) MRAMS

First we note that "shift left" instructions are unnecessary for VRAM acceptors: the proof is identical to the argument given to show that SUBSTR is not necessary for CRAMS -- roughly, that one shifts everybody else right instead of shifting one register left, and takes into account that initial segments of some registers should be considered garbage. With this in mind, all we have to simulate is the instruction

$$V_j + V_i \uparrow J_k \text{ (shift right)}$$

Clearly, this is equivalent to the MRAM instruction

$$V_j + V_i \cdot 2^{I_k}$$

and all we have to show is that it is possible

to have  $2^{I_k}$  in an MRAM register when  $I_k$  is used in a VRAM shift instruction. We shall argue, as in the CRAM case, that the contents of  $I_k$  cannot be too big: since the only operation that increases the contents of an index register is addition, the program that creates the biggest possible number in an index register in  $t$  steps of a VRAM's computation consists of adding a register repeatedly to itself. If the initial contents of the register was  $k$ , we produce  $2^t k$  after  $t$  operations. Therefore, in general, we must have at time  $t$  all index registers containing numbers of length  $t + c$  at most,  $c$  a constant depending only on the machine. But we may generate all numbers of the form  $2^m$ ,  $\text{length}(m) \leq t+c$  in time polynomial in  $t$ : we get the powers of 2 by multiplying 2 by itself and include those factors in the final product for which  $m$  has a 1 bit.

This proves the inclusion  $\text{PTIME} \supseteq \text{MRAM}$ .  $\text{PTIME} - \text{VRAM}$  and concludes the chain of implications proving  $\text{PTAPE} \subseteq \text{PTIME} - \text{MRAM}$ , the objective of this section.

#### 5. Conclusions and Comments

After the programming details of the previous two sections, it might be useful to restate the results of this paper. We defined a reasonable RAM model -- the MRAM -- that has multiplication as a primitive operation, and proved two important facts about their power as recognizers:

- 1) deterministic and nondeterministic time complexity classes are polynomially related (or  $\text{PTIME} - \text{MRAM} = \text{NPTIME} - \text{MRAM}$ ).
- 2) time-bounded computations are polynomially related to  $T_m$  tape (i.e.  $\text{PTIME} - \text{MRAM} = \text{PTAPE}$ ).

Since it can be proven that RAM time and  $T_m$  time are polynomially related, we also proved

- 3) RAM running times with and without multiplication are polynomially related if and only if  $T_m$  time and tape measures are polynomially related, i.e.  $\text{PTIME} = \text{PTAPE}$  iff  $\text{PTIME} - \text{MRAM} = \text{PTIME} - \text{RAM}_1$ .

This last observation is interesting, since it seems to imply that the elusive difference between time and memory measures for  $T_m$ s might perhaps be attacked by "algebraic" techniques developed in "low level" complexity theory. We obtained no results in this

direction: the sort of problem for which lower bounds on the number of multiplications are known compute functions, and for transducers we do already know that tape is more powerful than time.

We also note that RAMs may simulate MRAMs in polynomial time, as long as MRAMs operate in polynomial space and time. Therefore MRAMs are more powerful than RAMs if and only if the unit and logarithmic time measures are not polynomially related--i.e. if (in our "polynomial smearing" language) the two are distinct measures.

Many "if and only if" type categories follow, in the same vein, from 1), 2) and 3). For example:

"The set of regular expressions whose complements are non-empty ([1 Ch. 11]) is accepted in polynomial time by a deterministic  $T_m$  iff every language recognized by an MRAM in polynomial time is recognized by a deterministic RAM in polynomial time."

The reader may write down many of these: some of them sound quite surprising at first.

As we saw, if MRAMs are different from RAMs, they must use more than a polynomial amount of storage (in our simulation it was an exponential amount). This suggests asking whether it is sufficient to have a RAM<sub>1</sub> and exponential tape to get an MRAM's power, or, equivalently, to look at operations that make RAM - PTIME classes equivalent to PTAPE. The answer, as we saw, is that almost anything that expands the length of registers fast enough will do, as long as we have parallel bit operations: multiplication, concatenation or shifting all have this property. In particular, one of our CRAM models has nothing but concatenation, tests and parallel bit operations (no indirect addressing). On the other hand, we saw that adding more and more powerful operations (indirect addressing, shifts by shift registers, division by 2, SUBSTR, multiplication, integer division) do not make the model more powerful, once we have a fast memory-augmenting device. The stability of this class of RAMs makes them a nice characterization of memory-bound complexity classes. We also think they might be useful for studying parallelism.

Minsky suggested [7] that one of the objectives of theoretical computer science should be the study of trade-offs (e.g. between memory and time, nondeterminism and time etc.). Our constructions trade exponential storage for polynomial time (simulation of  $T_m$ s by MRAMs) and polynomial tape for exponential time in the other simulation. Whether this trade-off is real or the result of bad programming is not known, since  $P = \text{PTAPE}$ ? is an open problem. If  $P \neq \text{PTAPE}$ , then PTAPE would provide us with a class of languages which have a trade-off property: they may be recognized either in polynomial time or in polynomial storage, but not simultaneously.

Corollary:  $\text{PTIME} \neq \text{PTAPE}$  iff there exists a language  $L$  which can be recognized by MRAMs in polynomial time and polynomial memory, but not simultaneously.

Note that if such an  $L$  exists, any tape complete problem may be chosen to be it, for example  $L = \{R \mid R \text{ a restricted regular expression, } R = \Sigma^*\}$  [6].

We finalize by exhibiting a hierarchy of well-known problems in terms of restricted RAMs.

Lemma: The class of languages recognized by the restricted RAMs below are (instructions in brackets may be removed):

instruction set	restriction	language
<u>comp</u> , ( $\div$ ), ( $\pm$ ), <u>bool</u> , (i.a. <sup>†</sup> ), +, .	(N)PTIME	PTAPE
<u>comp</u> , ( $\div$ ), ( $\pm$ ), ( <u>bool</u> ), (i.a.), +	NPTIME	NP
<u>comp</u> , ( $\div$ ), ( $\pm$ ), ( <u>bool</u> ), (i.a.), +	PTIME	P
<u>comp</u> , ( $\div$ ), $\pm$ , <u>bool</u>	none	$\text{csl}^{\dagger\dagger}$
<u>comp</u> , ( $\div$ ), $\pm$ , <u>bool</u>	deterministic	$\text{dcs1}$
<u>comp</u> , ( $\div$ ), ( $\pm$ ), <u>bool</u> , (i.a.), +, .	$\text{time} < (\log n)^i$	$\text{LG}^{\dagger\dagger\dagger}$
<u>comp</u> , $\div$	division only by 2	NLOG-TAPE
<u>comp</u> , $\div$	division only by 2, deterministic	DLOG-TAPE
<u>comp</u> , $\div$	2 registers only	Regular sets

Lines 4 and 5 were observed by [12]; line 6 is obtained by noting that Theorems 1 and 2 hold when strengthened to time and tape constructible bounds greater than the logarithm; 7 and 8 follow from the characterization of logarithmic space-recognizable languages as the ones accepted by  $k$ -head finite automata for some  $k$ .

<sup>†</sup> indirect addressing

<sup>††</sup> context sensitive languages

<sup>†††</sup>  $\text{LG} = \{\text{languages recognizable in tape } (\log n)^i \text{ for some } i\}$

#### Bibliography

- [1] Aho, A., J.E. Hopcroft and J.D. Ullman: The design and analysis of computer algorithms. Addison-Wesley, Reading, Mass. 1974.
- [2] Cobham, A.: The intrinsic computational difficulty of functions. Proc. Inter. Conf. on Logic, Philosophy and Methodology of Science, Y. Bar Hillel ed. (North Holland, Amsterdam 1965) pp. 24-30.
- [3] Cook, S.: Linear time simulation of deterministic two-way pushdown automata. Information Processing 71 (North Holland Amsterdam, 1972) pp. 75-80.
- [4] Cook, S.: The complexity of theorem-proving procedures. Proc. 3rd Ann. ACM Symp. Theory Comp. pp. 151-158.
- [5] Cook, S. and S.O. Aanderaa: On the minimum computation time of functions. Trans AMS v. 142 (1969) pp. 291-314.

- [6] Hartmanis, J. and H. Hunt: The lba problem and its importance in the theory of computing. TR 73-171 Dept. Comp. Sci. Cornell University (1973).
- [7] Minsky, M.: Form and content in computer science. JACM v. 17 n. 2 (1970) pp. 197-215.
- [8] Pratt, V., L. Stockmeyer and M.O. Rabin: A characterization of the power of vector machines. Proc. 6th Ann. ACM Symp. Theory Comp. pp. 122-134.
- [9] Savitch, W.J.: Relationships between non-deterministic and deterministic tape complexities. JCSS v. 4 n. 2 (1970) pp. 177-192.
- [10] Stearns, R.E., J. Hartmanis and P.M. Lewis: Hierarchies of memory limited computations. IEEE SWAT Conf. Record (1965) pp. 179-190.
- [11] Sheperdson, J.C. and H.E. Sturgis: Computability of recursive functions. JACM v. 10 (1963) pp. 217-225.
- [12] Warkentin, J.C. and P.C. Fischer: Predecessor machines and regressive functions. Proc. 4th Ann. ACM Symp. Theory Comp. pp. 81-87.

#### Acknowledgments

The authors would like to thank Dr. John Hopcroft and Zvi Galil for many useful discussions. In particular the algorithm for simulation of division by using multiplication in Section 3 was brought to our attention by Professor Hopcroft.

