stichting
mathematisch
centrum

MC

AFDELING INFORMATICA
(DEPARTMENT OF COMPUTER SCIENCE)

IW 193/82 FEBRUARI

P.M.B. VITÁNYI

REAL-TIME SIMULATION OF MULTICOUNTERS BY OBLIVIOUS ONE-TAPE TURING MACHINES
(PRELIMINARY DRAFT)

Preprint

kruislaan 413 1098 SJ amsterdam

Printed at the Mathematical Centre, 413 Kruislaan, Amsterdam.

The Mathematical Centre, founded the 11-th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

1980 Mathematics subject classification: 68C40, 68C25, 68C10

ACM-Computing Reviews-category:5.23, 5.25, 5.26

Real-time simulation of multicounters by oblivious one-tape Turing machines*)
(Preliminary draft)

by

Paul M.B. Vitányi

ABSTRACT

Each multicounter machine can be simulated in real-time by an oblivious one-head tape unit.

KEY WORDS & PHRASES: *Multicounters, real-time simulation, one-tape Turing machines, oblivious one-head tape unit, integer representation*

---

*) This report will be submitted for publication elsewhere.

To appear in the Proceedings of the Fourteenth Annual Symposium on Theory of Computing, held in San Fransisco, May 5-7, 1982

1. Introduction

In many computations it is necessary to maintain several counts simultaneously such that, at all times, an instant signal indicates which counts are zero. Keeping many counts in tally notation, each count being incremented/decremented independently by at most 1 in each step, governed by the input and the set of currently zero counts, is formalized in the notion of a multicounter machine [1,2]. Such machines have numerous connections with both theoretical issues and more or less practical applications. One of the better known open problems about counter machines concerns the question whether or not multicounter machines can be simulated in real-time by one-tape Turing machines [1,2]. The result in this paper answers the question affirmatively, with a considerable margin, since the exhibited one-head tape units real-time simulating the multicounter machines are oblivious. By well known constructions we find as a corollary that the first n steps of a multicounter machine can be implemented on a fast low-cost combinational logic network.

Multicounter machines. We view machines as *transducers*. (The simulation results therefore hold also for the machines viewed as recognizers.) Thus we abstract from the input/output conventions and concentrate on the storage structure. The abstract storage structure embodied by a k-*counter machine* (k-CM) consists of a finite state control unit, k *counters* each capable of containing any integer, and an input and output terminal. The states of the finite control are partitioned into *polling* and *autonomous* states. At the start of the computation the k-CM is in a designated initial state and the counters are set to zero. A *step* in a CM computation is uniquely determined by the state of the finite control, by the symbol scanned at the input terminal if the state is a polling state and by the set of counters which contain zero. The action at that step consists of independently altering the contents of each counter by adding -1, 0 or +1, changing the state of the finite control and producing an output. Hence the machine effects a transduction from input to output strings. If you will, the input and output might be written on some input and output tapes on which the access pointers (heads) are steered by the finite control. The above is more or less the formulation in [2] where also a more precise definition can be found.

Background of the problem. Counter machines are relatively old devices. Unrestricted 2-counter machines were shown to be as powerful as Turing machines in [4]. Subsequently the power of counter machines was measured against resource restricted multitape Turing machines. In [2] the now classic linear time/logarithmic space on-line simulation of multicounter machines by one-tape Turing machines was exhibited. Since then it has been an open problem whether or not k-1 single head tape units suffice to simulate a k-counter machine in real-time. Stated imprecisely, but perhaps more to the point, whether or not k counters can be maintained in real-time by a linear storage unit with less than k access pointers. For the easier Origin Crossing Problem, where the task is to recognize the set of sequences of unit basis vectors in k-space, which leave from and end in the origin, an ingeneous solution on a real-time one-tape Turing machine appeared in [1]. The Axis Crossing Problem, of recognizing the set of sequences of unit basis vectors in k-space which leave from the origin and end in one of the (k-1)-dimensional hyperplanes with one zero coordinate, defied a real-time (k-1)-tape Turing machine solution [1,2]. In [7] the linear time/logarithmic space one-tape solution for simulating multicounter machines was made oblivious retaining the same resource bounds. This is significant, since by its nature an oblivious multitape Turing machine is far more restricted than a non-oblivious one. For instance, an oblivious multitape Turing machine needs $\Omega(n \log n)$ steps to on-line simulate a single pushdown store, and a 2-tape oblivious Turing machine achieves this bound [5]. There is good reason to suppose that an oblivious one-tape Turing machine requires $\Theta(n^2)$ steps

for this task. With respect to counters, the story is different due to the possible compactification of information to be stored. In [6, Corollary 2] it was shown how to on-line simulate a T(n) time / S(n) storage multitape Turing machine by an oblivious 2-tape Turing machine in time $O(T(n) \log S(n))$ and storage $O(S(n))$. So the best simulation by an oblivious multitape Turing machine known before [7], was achieved by combining [2] and [6], yielding an on-line simulation of n steps of a multicounter machine by $O(n \log \log n)$ steps of an oblivious 2-tape Turing machine. The real-time simulation presented here gives a surprising example of the power of a one-head tape unit, let alone an oblivious one, besides shedding light on encodings for dynamically maintaining integers under unit incrementations/decrementations.

Obliviousness. A Turing machine is *oblivious*, if (for the moment disregarding the input/output structure) the movements of the storage heads are fixed functions of time independent of the inputs to the machine [5,6]. Many problems seem inherently oblivious: the usual algorithms for computing the four main arithmetic functions, or a table look-up by sequential search, can easily be programmed obliviously. Other problems like e.g. binary search are inherently nonoblivious. Reasons to consider oblivious computations have been that they are easily converted to combinational logic networks or straight line programs. Furthermore, when we restrict ourselves to this class of computations it becomes simpler to derive good lower bounds on time complexity, or good time/space trade offs, for certain problems to be solved on the chosen model of computation, whereas we consistently fail to do so in the nonoblivious case. This seems especially worthwhile for problems where it is hard to see how nonobliviousness could yield any gain, or where the change of a nonoblivious computation model to an oblivious one incurs a relatively small penalty [5,6]. Here we show yet another, more heuristic, motive for confining attention to oblivious computation: restriction of the considered model of computation to its oblivious version can shift the emphasis in the problem to be solved from one aspect to another one directing us to a solution. Thus, we here use the oblivious restriction to obtain upper bounds on complexity. Whereas the problem of real-time simulating k-counter machines by k'-tape Turing machines, $k' < k$, is solely due to the fact that $k' < k$, the same problem for oblivious Turing machines knows only one difficulty: the obliviousness of the simulating device. For suppose we can on-line simulate n steps of some abstract storage device S in T(n) steps by an oblivious Turing machine M. Then we can also simulate a collection of k such devices $S_1,\ldots,S_k$, interacting through a common finite control, by dividing all storage tapes of M into k tracks each of which is a duplicate of the corresponding former tape. The same head movements of M can do the same job on each of the k collections of tracks as formerly on the collection of tapes. So the time and storage complexity of the extended M is the same as those of the original. By these scant considerations we obtain in particular:

Proposition 1. *The following statements are equivalent:*

(i) *We can simulate each* 1*-counter machine by an oblivious* 1*-tape Turing machine in real-time.*

(ii) *For each* k*, we can simulate each* k*-counter machine by an oblivious* 1*-tape Turing machine in real-time.*

Approach to the problem. It quickly becomes apparent that updating a count correctly in real-time on an oblivious machine requires a redundancy in notation which seems to make an instant check for zero impossible. To achieve the latter, we shall allow only encodings of integers such that they are zero iff the 'first' positions show this uniquely. Since the head movement is supposed to be oblivious we must, roughly speaking, update each 'initial' log i length segment of the encoded integer within each cycle of about i steps. This entails that, while moving the head to update a longer segment of code, we have to simultaneously shift and update smaller segments of code, and so on recursively down to the smallest pieces.

Outline of the simulation. Like in [2] we maintain the current count c as the difference between two positive integers y and z. Like in [7] we encode y and z in a redundant binary notation using digits 0,1,2, where a 2 in a bit position stands for an unprocessed carry. Leading 0's in corresponding positions in y and z are immediately replaced by a distinguished blank symbol. Maintaining also an invariant such that if any position of either integer y,z contains a digit greater than 0 then the three nearest positions of the other integer do not, ensures that $y - z = 0$ iff y and z contain blanks only. Incrementing [decrementing] the count by 1 is done by incrementing y[z] by 1. In order to be able to increment y[z] at all times by 1, and to also maintain said invariant, we continuously subtract corresponding blocks of digits in y and z from each other. Although there is a considerable amount of freedom here how to go further, for reasons of exposition we choose to divide the encoding of y and z into consecutive blocks of 10 digits (in corresponding positions) each, and update the i-th block within each cycle of 2^{10i} steps. For this purpose, we develop a method to recursively transport the digits of block i+1 from one side of the combination of the first i blocks to the other side, back and forth, one digit within each cycle of 24^i steps, for all i. The process shall be such that the single head, without being able to identify individual pieces of encoding, still maintains a topological adjacency relation amongst them, which allows it, according to certain local criteria, to update correctly. For the recursive process we use an extra pushdown store and also allow squares on the tape to be deleted and inserted. We later get rid of the square deletion/insertion option by interleaving a stack of deleted squares through the sequence of consecutive blocks, which by the recursive block traversal can grow and shrink as required. Similarly, we maintain the pushdown store. The net effect of all this will be that, for all i, the combination of the first i blocks acts like a single very fat head, moving slower the higher i is, but fast enough to update block i+1 (plus a few adjacent positions which will be required) each small enough cycle of steps.

The present paper is a preliminary and incomplete draft; it does however contain all of the simulation. A more extensive version containing all necessary proofs will appear later. For notions like on-line simulation, real-time (simulation) etc., see e.g. [5].

2. Real-time simulation of a counter by an oblivious one-head tape unit

By Proposition 1, we can simulate each multi-counter machine in real-time by an oblivious one-head tape unit if we can simulate a single counter that way. So let C be a single counter, and let S be an abstract linear ordered storage unit, for simulating C.

2.1. Notation of the count

The count c is maintained as difference between two nonnegative integers y and z. If at the current step of C $\delta \in \{-1,0,+1\}$ is added to the count, then

$\delta = -1$: $c \leftarrow c-1$ $y \leftarrow y$ $z \leftarrow z+1$
$\delta = 0$: $c \leftarrow c$ $y \leftarrow y$ $z \leftarrow z$
$\delta = +1$: $c \leftarrow c+1$ $y \leftarrow y+1$ $z \leftarrow z$.

We also allow at any time to subtract equal amounts from y and z such that both y and z stay nonnegative integers. So, for all t, at the t-th step of C (and of the simulator S) the following invariants will hold:

(1) $y + z \leq t$
(2) $y - z = c$.

Both y and z are represented in a kind of *redundant* binary notation: $code(y) = y_0y_1\ldots y_m$ and $code(z) = z_0z_1\ldots z_m$, $y_i,z_i \in \{0,1,2\}$, $0 \leq i \leq m$, such that

$$y = \sum_{i=0}^{m} y_i 2^i \quad \text{and} \quad z = \sum_{i=0}^{m} z_i 2^i$$

and

$$m = \max \{i \mid y_i > 0 \vee z_i > 0\}.$$

In the simulation we maintain y and z on two channels: the y- and z-channel, and replace nonsignificant 0's in corresponding digit positions on both channels simultaneously by blanks. At all times t we maintain the additional invariant:

(3a) $(y_i > 0 \Rightarrow z_{i-1},z_i,z_{i+1} \in \{0, \text{blank}\})$ &
$(z_i > 0 \Rightarrow y_{i-1},y_i,y_{i+1} \in \{0, \text{blank}\})$ &

(3b) $\neg(y_i = z_i = 0 \;\&\; y_{i+1} = z_{i+1} = \text{blank})$ &
$(y_i = \text{blank} \Longleftrightarrow z_i = \text{blank})$

with the obvious allowances made for the borderline case i = 0.
(3a) ensures that if the i-th digit position of code(y) contains a digit greater than 0 then the positions i-1, i, i+1 of code(z) contain no digits greater than 0 and vice versa.
(3b) takes care that there occur no leading 0's in corresponding positions of code(y) and code(z).

Although not only the encoding of c as the difference of y and z is not unique, but also the encoding of y and z themselves are not unique, we still have:

Proposition 2. *Since invariants* (1) - (3) *hold at each step of the simulation,* c = 0 *iff code*(y) = = *code*(z) = ε.

Proof. For suppose there is a 0,1 or 2 in either code(y) or code(z). Then according to (3) there is also a highest indexed y_m or z_m greater than 0, say $y_m > 0$. But then, by (3) $z \leq 2 \sum_{i=0}^{m-2} 2^i \leq 2^m-2$. Since $y_m > 0$ we have $y \geq 2^m$ and by (2) it follows $c = y - z \geq 2$. (For m = 0 or m = 1 allow for the borderline case.) Conversely, if $c \neq 0$ then code(y), code(z) $\neq \varepsilon$ by (2) and (3). □

2.2. Maintenance of the count.

For the moment we think of y and z as being maintained on a tape divided into 2 channels: the y- and z-channel. Code (y) and code(z) are justified with the low order digits in the home square 0. The tape is one-way infinite, and squares m+1, m+2,... contain blanks denoted by "-", where m is highest index of a non-zero digit in code(y) or code(z). See Figure 1.

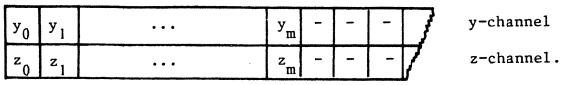


Figure 1.

The tape is divided into blocks of length x, say x = 10, which are called block 0, block 1,..., see Figure 2.

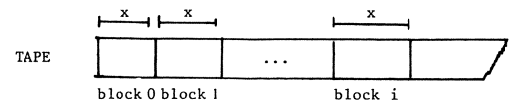


Figure 2.

During the real-time simulation of C by S we shall update each block i together with the first two positions of block i+1 at least once within each cycle of T(i) steps, such that T(0) = 1 and $T(i+1) < 2^{10} T(i)$ for all $i \geq 0$. To describe the updating procedure, and to show that it maintains invariants (1) - (3) throughout the entire tape, we consider the digit positions involved in the updating of a block i in isolation, and divide them into an *input field* of 2 digits, followed by an *own field* of 8 digits, followed by an *output field* of 2 digits. The digit position next to the output field will serve to *check* for adjacent blanks so as to remove leading 0's. See Figure 3.

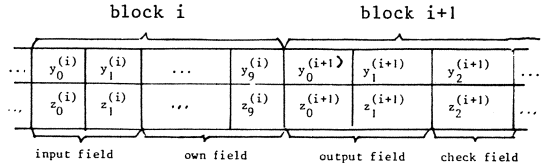


Figure 3.

So the input field of block i is the output field of block i-1, i > 0, while the output field of block i is the input field of block i+1, $i \geq 0$. For the sake of uniformity, we consider the processing of the current input to be the updating of a hypothetical block -1, of which the output field is the input field of block 0. If at any step more than 1 block needs to be updated, we update the higher indexed blocks first. Note that since an update of block i will involve also its output field two consecutive blocks i and i+1 are concerned.

Let $\begin{matrix} y_0^{(i)} y_1^{(i)} \ldots y_9^{(i)} \\ z_0^{(i)} z_1^{(i)} \ldots z_9^{(i)} \end{matrix}$ be the *current contents* CC(i) of block i, $i \geq -1$, and let the *local count* on block i be $c_i = Y_i - Z_i$, where $Y_i = \sum_{j=0}^{9} y_j^{(i)} 2^j$ and $Z_i = \sum_{j=0}^{9} z_j^{(i)} 2^j$. Let the current input be $\delta \in \{-1,0,+1\}$ and let blocks $i_1, i_2, \ldots, i_m$, $i_1 < i_2 < \ldots < i_m$, be the blocks to be updated by UPDATE below in the current step. (It will appear that always $i_1 = -1$, $i_2 = 0$, while $m \leq 4$). Assume that all nonblank symbols on the tape are put there by previous applications of UPDATE, and that UPDATE preserves invariants (1) - (3). This assumption will be justified, since (1) - (3) hold at the start and, under the timing assumptions in the next Proposition 3, UPDATE preserves (1) - (3).

Algorithm UPDATE:

step 1. $c_{-1} \leftarrow \delta.2^{10}$; $i \leftarrow i_m$; $k \leftarrow m$;

step 2. Select the appropriate case below and execute the action.

0. $Y_i = Z_i$ → nothing changes.

1. $Y_i > Z_i$

1.1. $Y_i - Z_i < 2^9$

1.1.1. $Y_i - Z_i$ is even → CC(i) $\leftarrow$ $\begin{matrix} 0 y_1 y_2 \cdots y_8 0 \\ 00\,0 \ldots 0\,0 \end{matrix}$ with $\sum_{j=1}^{8} y_j 2^j = Y_i - Z_i$ and $y_j \in \{0,1\}$, $1 \leq j \leq 8$.

1.1.2. $Y_i - Z_i$ is odd → ¢ By invariant (3a) there is no '2' in the first position on block i: nothing changes ¢

1.2. $Y_i - Z_i = 2^9$

1.2.1. $z_0^{(i+1)} > 0$ → $z_0^{(i+1)} \leftarrow z_0^{(i+1)} - 1$; CC(i) $\leftarrow$ $\begin{matrix} 00\ldots00 \\ 00\ldots01 \end{matrix}$.

1.2.2. $z_0^{(i+1)} \neq 0$ → CC(i) $\leftarrow$ $\begin{matrix} 00\ldots01 \\ 00\ldots00 \end{matrix}$.

1.3. $2^9 < Y_i - Z_i < 2^{10}$ ¢ UPDATE can deposit '2''s *only* in the first square of a block, so $y_9^{(i)} = 1$. By invariant (3a) therefore $z_0^{(i+1)} \in \{0,-\}$. So similar to case 1.1: ¢

1.3.1. $Y_i - Z_i$ is even → CC(i) $\leftarrow$ $\begin{matrix} 0 y_1 y_2 \ldots y_8 y_9 \\ 00\,0 \ldots 0\,0 \end{matrix}$ with $\sum_{j=1}^{9} y_j 2^j = Y_i - Z_i$ and $y_j \in \{0,1\}$, $1 \leq j \leq 9$.

1.3.2. $Y_i - Z_i$ is odd → ¢ By invariant (3a) there is no '2' in the first position on block i: nothing changes ¢

1.4. $Y_i - Z_i = 2^{10}$ → CC(i) $\leftarrow$ $\begin{matrix} 00\ldots0 \\ 00\ldots0 \end{matrix}$; ¢ and according to the appropriate subcase the output field $\begin{matrix} y_0^{(i+1)} y_1^{(i+1)} \\ z_0^{(i+1)} z_1^{(i+1)} \end{matrix}$ is updated. ¢

1.4.1. $y_0^{(i+1)}, z_0^{(i+1)}, z_1^{(i+1)} \in \{0,-\}$ → $y_0^{(i+1)} \leftarrow 1$; $z_0^{(i+1)} \leftarrow 0$.

1.4.2. $y_0^{(i+1)} = 1$ → $y_0^{(i+1)} \leftarrow 2$.

1.4.3. $z_0^{(i+1)} = 1,2$ → ¢ By invariant (3a) impossible. Viz. UPDATE can deposit '2''s *only* in the first square of a block, so $y_9^{(i)}$ was 1. ¢

1.4.4. $z_1^{(i+1)} = 1$ → $z_0^{(i+1)} \leftarrow 1$; $z_1^{(i+1)} \leftarrow 0$.

2. $Y_i < Z_i$. ¢ Analogous to case 1 with the roles of the y- and z-channels interchanged ¢.

step 3. Inspect $y_2^{(i+1)}$ to see whether it is a blank '-'. If so, replace all leading 0's by blanks in block i and its output field up to the leftmost position of a nonzero digit.

step 4. If i = -1 then finish else(i $\leftarrow$ i_{k-1}; k $\leftarrow$ k-1;goto step 2).

UPDATE simply removes a 2 from the first square of a block i and propagates it right, until it peters out or arrives at the first square of block i+1, meanwhile preserving invariant (3) throughout the concerned tape segments and invariant (1) and (2) by the update of block -1, treating the higher indexed blocks first.

Proposition 3. *If each block* i, $i \geq -1$, *with the stated interpretation of 'updating block -1' as processing the current input, is updated by UPDATE at least once in every cycle of* T(i) *steps of* S, T(-1) = T(0) = 1 *and* $T(i+1) < 2^{10}\, T(i)$ *for* $i \geq 0$, *then UPDATE always succeeds and invariants* (1)-(3) *are preserved.*

Proof. By induction on the number of steps. *Base case* t = 0. Initially all positions on the y- and z-channel contain only blanks and so (1) - (3) hold.

Induction phase. Assume that the proposition held up to time t: so UPDATE has always succeeded and invariants (1) - (3) hold at time t. Let blocks $i_1, i_2, \ldots, i_m$ be due for updating at the current step, $i_1 < i_2 < \ldots < i_m$, $i_1 = -1$, $i_2 = 0$. Since UPDATE never leaves a 2 anywhere on the tape except possibly in the first position of a block, and also preserves invariants (1) - (3), this is the condition of the tape at time t. We need to show that under the assumptions of the proposition this is also the case at time t+1. The due blocks $i_1, i_2, \ldots, i_m$ at the current step are updated in order $i_m, i_{m-1}, \ldots, i_1$ by UPDATE. Since UPDATE has succeeded all previous steps, it possibly has put a 2 in the first position of a channel on a block but nowhere else, and we have at time t that $|c_i| \leq 2^{10}$ for all $i \geq 0$. The only way UPDATE can fail on block i_m is for $c_{i_m} = 2^{10}$ and $y_0^{(i_m+1)} = 2$ or $c_{i_m} = -2^{10}$ and $z_0^{(i_m+1)} = 2$. Assume that the former is the case. According to the rules of UPDATE $y_0^{(i_m+1)}$ must have been incremented to 2 at an earlier UPDATE of block i_m, say at the step from t'-1 to t'. So block i_m contained only 0's at time t' according to 1.4.2, and D = (the total count on the y-channel minus the total count on the z-channel on the consecutive blocks $0,1,\ldots,i_m$) satisfies $|D| \leq \sum_{i=1}^{i_m} 2^{10i}$. Since $T(i_m+1) \leq (2^{10}-1)^{i_m+1}$ by hypothesis in the proposition, we have $t-t' \leq (2^{10}-1)^{i_m+1}$. Otherwise the '2' in the first position on block i_m+1 would have disappeared by a successfull application of UPDATE to that block in the meantime. Hence at time t, $|D| \leq \sum_{i=1}^{i_m} 2^{10i} + (2^{10}-1)^{i_m+1}$, because the '2' has stayed in the first position on block i_m+1's y-channel, and hence by invariant (3) and the rules in UPDATE no digits have been transported from block i_m+1 to block i_m in the meantime. We saw that for UPDATE not to succeed on block i_m at time t, $c_{i_m} = 2^{10}$, and so $|D| \geq 2^{10(i_m+1)}$. Since this leads to the contradiction

$$2^{10(i_m+1)} \leq \sum_{i=1}^{i_m} 2^{10i} + (2^{10}-1)^{i_m+1}$$

for all $i_m \geq 0$, UPDATE succeeds on block i_m. It is easy to see that UPDATE maintains invariant (3a) on block i_m in between the first position on its input field and the last position on its output field. So we only need to worry about the maintenance of (3a) on the interfaces between blocks i_m-1 and i_m

and blocks i_m and i_m+1. However, either the entries in block i_m and its output field are not changed at all (rules 0,1.1.2 and 1.3.2) or no 1's and 2's are left at all in the first position of the input field, preserving (3a) locally on the interface of blocks i_m-1 and i_m, and no 1's and 2's are left in the output field in positions where they might violate (3a), assuming invariant (3a) held before the update, preserving (3a) locally on the interface between blocks i_m and i_m+1 (check rules 1.1.1, 1.2, 1.3.1, 1.4). Thus, step 2 of UPDATE preserves invariant (3a). During the update of block i_m, $y_2^{(i_m+1)}$ and $z_2^{(i_m+1)}$ were not changed. So either they were not blank and no leading 0's were created by step 2 of UPDATE, assuming invariant (3b) held previously, or they were blank and all new leading 0's created by step 2 are turned into blanks by step 3. The only remaining possibility to fail maintaining invariant (3b) is on the interface between blocks i_m-1 and i_m, viz. by leaving block i_m with blanks only while block i_m-1 has $y_9^{(i_m-1)} = z_9^{(i_m-1)} = 0$, and block i_m did not contain only blanks before step 3. But this last case is excluded because either $Y_i = Z_i$ ($i \leftarrow i_m$) and step 2 didn't change anything (rule 0) or $Y_i - Z_i$ was odd and nothing changed (rules 1.1.2 and 1.3.2), or step 2 did not leave leading 0's at all in the first position of the input field of block i_m (rules 1.1.1, 1.2, 1.3.1, 1.4). Therefore step 2 followed by step 3 of UPDATE maintain invariant (3) throughout the tape. Similarly, the subsequent due updates of blocks $i_{m-1},\ldots,i_2$ succeed. Since $T(0) = 1$, the update of block -1 (i.e., the processing of the input) always succeeds during the execution of UPDATE, because the input field of block 0 contains no '2''s when block -1 is updated. So the successive updating of blocks $i_m, i_{m-1},\ldots,i_1$ during the execution of UPDATE succeeds. Hence the current input is processed by UPDATE, maintaining invariants (1) - (3). So, under the timing assumptions in the Proposition, at each step UPDATE succeeds it preserves invariants (1) - (3), while under the assumption that invariants (1) - (3) hold at time t by applications of UPDATE, starting from a blank tape, UPDATE succeeds in the next step. □

One may well wonder how it is possible that while at most 4 blocks, so less than 50 positions, are updated each step, invariant (3) can be maintained, since it allows the tape to contain

$$\underbrace{00\ldots001-}_{\text{length } m}$$
$$22\ldots200-\ ,$$

so $c = y - z = 2$, and c can be set to 0 in 2 steps.

We have, however, excluded such racing of the most significant digit to the home position by implicitly maintaining a far stronger invariant than (3a) in step 2 of UPDATE, viz.

(3a')

$$\forall i \geq 0[\exists j \in \{1,\ldots,9\}[y_j^{(i)} > 0] \Rightarrow \forall j \in \{1,\ldots,9\}[z_j^{(i)} \in \{0,-\}]] \ \&$$

$$\forall i \geq 0[\exists j \in \{1,\ldots,9\}[z_j^{(i)} > 0] \Rightarrow \forall j \in \{1,\ldots,9\}[y_j^{(i)} \in \{0,-\}]] \ \&$$

$$\forall i \geq 0\ \forall j \in \{1,\ldots,9\}[y_j^{(i)} \neq 2] \ \& \ (3a).$$

The present proof relies partly on the fact that all of the nonblank tape contents at any time must be the result of previous applications of UPDATE alone, and partly on invariant (3a).

2.3. The oblivious one-head tape unit

According to Propositions 2 and 3 we can simulate a counter C in real-time by an abstract storage unit $\mathcal{S}$ consisting of a finite-state control and a linear ordered set of storage locations called *cells*, which are in one-to-one correspondence with the tapesquares of the previous section (using the cells as boxes in which the contents of the corresponding squares are stored), provided the following Requirement is met.

Requirement. For all $i \geq 0$, within each cycle of $T(i)$ steps, $T(0) = 1$ and $T(i+1) < 2^{10} T(i)$ for all $i \geq 0$, there is a time instant at which S accesses the ordered group of cells corresponding to the squares of blocks i and i+1. Whenever S accesses more such groups of cells simultaneously in some time instant, it distinguishes the relative order amongst those groups, induced by the corresponding block indexes.

If S meets the Requirement, then S can each step execute UPDATE meeting the timing conditions in Proposition 3, on the appropriate square contents as contained in the corresponding groups of cells, thus maintaining invariants (1) - (3). Since $T(0) = 1$, S always knows whether block 0 contains only blanks, and so by Proposition 2 whether the current count is zero or not.

Below we describe an oblivious one-head tape unit supporting such an abstract storage unit S capable of simulating each counter C in real-time. Thus, by Proposition 1, an oblivious one-head tape unit can simulate each multicounter machine in real-time.

Let M be a one-head tape unit consisting of a finite control, a one-way infinite storage tape divided into squares, and a single read-write head on the tape covering $2\ell+1$ squares, for ℓ a constant (ℓ=25 will suffice). Originally the head covers the left $2\ell+1$ squares on the tape. By the *position* of the head we shall mean the position of the central square covered, so the start position is the $(\ell+1)$-th square from the left. During its action, by marking newly met unmarked right adjacent squares, and by counting modulo 10 in its finite control, M will lay out the blocks of 10 squares each on its tape, beginning at the start position. The cells corresponding to block 0 are maintained in M's finite control, and the cells corresponding to block i, $i > 0$, of the previous section can be thought of as being contained initially by the corresponding squares of the i-th block to be determined on M's tape. Initially all squares on M's tape contain only blanks, and so do the cells contained on those squares. M creates such an appropriate cell whenever it meets an unmarked square on its tape, marking it as a left border '[' if it is the first cell of a new block, as an internal cell 'o' if it is not a border cell, and as a right border ']' if it is the last cell of a block. Finally, M distinguishes the cells of block 2 from those of the other blocks by either tagging them initially or remembering where they are (they will always be covered by M's head). So the start situation can be viewed as depicted in Figure 4.

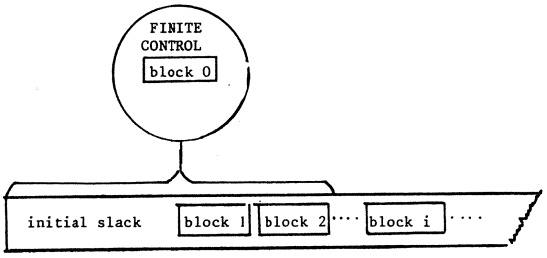


Figure 4.

Except for the determination of new blocks, the identity of the actual squares on M's tape is not important, but the identity of the created cells is fixed wherever they end up in the simulation. So in the sequel when we talk about a block we will mean a block of cells. The head is forever positioned on block 1 and executes a regular motion from the leftmost position on block 1 to the rightmost position and back to the leftmost position, one square each step. At the extremes of its sweeps it therefore scanns alternatively ℓ squares left of block 1 and ℓ squares right of block 1. For the moment we assume that M can delete a square adjacent to block 1 and insert a square adjacent to block 1. We also assume that we have an extra pushdown store available. Later we show how to eliminate these options.

At all times blocks 1,2,...,i shall form a 'topologically connected' segment 'adjacent' to block i+1 for all i ≥ 1. We shall call this segment *superblock* I. Roughly, the following scheme is recursively executed for all superblocks I:

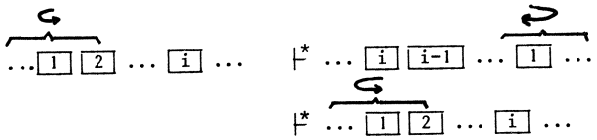


Figure 5

We describe loosely what happens at first and then supply more detail. Initially the head is centered on the leftmost square of block 1. It then moves right to the rightmost square, scans the leftmost border of block 2, stores it on the pushdown store, moves to the leftmost square of block 1 and inserts the leftmost border (on top of the pushdown store) next to the leftmost border of block 1, replacing the instruction on top of the pushdown store by an instruction to fetch a new cell. Squares are deleted/inserted as needed. (See Figure 6).

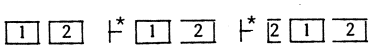


Figure 6

In the next 8 sweeps the head moves to the right end of block 1, stores the cell contents of the right adjacent square of block 2 on the pushdown store, deletes the square, moves to the left end of block 1, inserts a square and places the stored cell of block 2 in it and replaces the top of the pushdown store with a fetch instruction, in each sweep. The resulting situation is depicted in Figure 7.

Figure 7

When the head in its rightmost position scans the right border of block 2 and the left border of block 3, the left border of block 3 is placed on the pushdown store, the action is reversed and block 1 traverses block 2 in the other direction until only the left border of block 2 remains, inserts the left border of block 3 left of that of block 2, and traverses block 2 again up to and including the right boundary of block 2. The resulting situation is depicted in Figure 8.

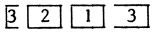
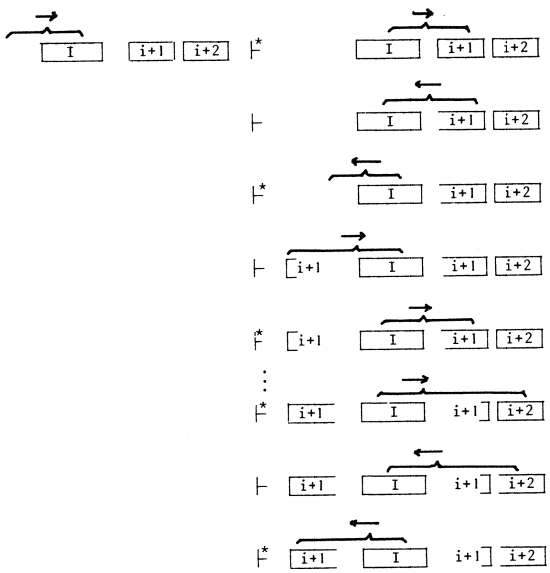


Figure 8

Now the combination of blocks 1 and 2 starts to traverse block 3, beginning with storing the first internal cell of block 3 on the pushdown store and deleting the corresponding square, then traversing block 2 by block 1 completely up to and including the left border of block 2 (which is not now adjacent to a differently oriented border of some block) inserting the cell now on top of the pushdown store on a newly inserted square and so on. The general scheme of how the first i blocks act with respect to block i+1 is shown in Figure 9.

The union of blocks 1,2,...,i is *superblock* I. Cells deleted from the tape are pushed on the pushdown store, and cells inserted are popped from the pushdown store. The pushdown store is not displayed below, and the head scans at least ℓ-1 elements next to block I in the displayed positions. (Since we shall set ℓ=25 the head scanns more cells than drawn). Squares and cells are deleted/inserted in concert.



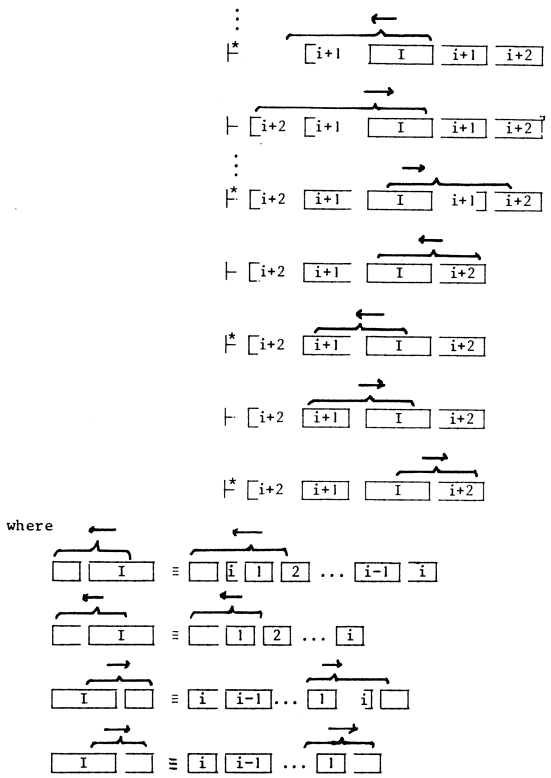
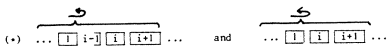


Figure 9

We are able to do this data transport by having the head scan the two adjacent squares when it is at the right or left border of block 1, and inspecting the top of the pushdown store. We only need to distinguish elements in left borders '[', right borders ']' and internal cells; the transition rules are designed so that we do not have to know to which block the elements belong. Automatically, by the maintenance of the pushdown store and a loose binding of '][' pairs, a certain order of blocks is preserved. In general, the cells within a block are always in the correct left to right order, but a sequence of blocks is in the correct left to right order iff all blocks of the sequence are "closed" and on the right of block 1, where block 2 is identified by marked borders. (The borders of block 2 are the only ones which are identifiable as belonging to a particular block).

The effect of this procedure is that, like a very fat head superblock I continuously traverses block i+1 from left-to-right and right-to-left. The higher index i is the slower this goes. We now estimate whether this allows us to meet the Requirement on S. To later eliminate the square insertion/deletion facility, we need to be able to expand each block by two squares (without cells). Since block 1 is a special case it will need to store at most three such squares. In the following we shall call a block *closed* if all of its cells are present on the tape in a connected segment, disregarding intermittent squares containing no cells. A block is *open* if it is not closed. In situations like



the head in its most extreme right position on block 1 scanns 2 consecutive *closed* blocks right of block 1 and therefore will know that these blocks are blocks i and i+1 for some i > 1. Since the relative order of cells within a block, of the cells present on the tape, is never changed by the cell transport, the cells of blocks i and i+1 are consecutive and in the correct left-to-right order. In the leftmost situation of (*) i > 2, and in the right situation of (*) i ≥ 2. Since the machine always knows which block is block 2 (by the tagging of its elements), the blocks due for update in (*) left are i, 0 and -1 (processing of input) and in (*) right they are blocks i, 0 and -1 if i > 2 and i,1,0,-1 if i=2. Note that M doesn't need to know the indexes of the blocks, except that of block 2 in case block 1 is updated, to update the due blocks highest indexes first.
Let S(i) be the number of steps needed for a full sweep by the head over superblock I, starting when it has just deleted or inserted (or was in the initial situation) a square left of superblock I, going right until it insert/deletes a square right of superblock I and back up to, and including, the insertion/deletion of a square left of superblock I. Clearly S(1) ≤ 24. Since each block i, i > 1, covers at most 12 squares, not more than 24 full sweeps over superblock I-1 are required to make one full sweep over superblock I, and we have S(i) ≤ 24S(i-1). Hence $S(i) \leq 24^i$ for all i ≥ 1. In each full sweep over superblock I+2 configuration (*) must occur at least once, see figure 9. We can choose T(i) so that

$$S(i+2) \leq 24^{i+2} < T(i) \leq (2^{10}-1)^i$$

for all i ≥ 2. Block 0 is contained in the finite control, while by setting $\ell = 25$ superblock 2 is always completely covered on the tape, so we can choose T(0) = T(1) = 1. Hence an abstract storage unit S meeting the Requirement is supported by the loosely described one-head tape unit M. We describe M in detail by listing the rules governing the device. Squares containing left and right border cells of blocks are denoted by '[' ']' respectively. Squares containing any internal cell of a block, or no cell at all to later accommodate the square deletion/insertion facility, are denoted by 'o'.
The format of the rules is:

(a,b,c) ⇒ (d,e,f), where

a = direction of last move.
b = scanned outside block 1 in direction of last move.
c = top of stack.
d = direction of new move.
e = string replacing b on the tape.
f = 0,1 or 2 elements c is replaced with (top of stack is right)

The stack alphabet consists of:

(*a*) fetch
(*b*) store <operand>

(*c*) store <operand> ^ fetch
(*d*) store <operand> ^ store <operand>

with <operand> ∈ {o,[,]}

(*c*) and (*d*) indicate two 'linked' stack instructions.

- If the head is positioned on an internal cell of block 1, b and e are NIL.
- Since the machine has special rules when it scans a 'closed' block 2 right or left of block 1, we write '2' for 'o' in b and e in such cases.

(right, o, fetch) ⇒ (left, ε, store o) means: the head moving right scans, right of the central square scanned, containing the right border of block 1, an adjacent square containing an internal cell. The top of the pushdown store contains a 'fetch' instruction. In one step it does all of the following:

(i) Replace the top of the pushdown stack by 'store o'
(ii) delete the square containing 'o'
(iii) move the head 1 square left.

As long as the center scanned square contains an internal cell 'o' the rules specify (as stated before)

R0. (right, NIL, top) ⇒ (right, NIL, top)

L0. (left, NIL, top) ⇒ (left, NIL, top)

when 'top' stands for current top of the stack. The remainder of the rules is as follows:
RULES:
R1. (right,[2,top) ⇒ (left,2,top store[),
R2. (right,][,fetch) ⇒ (left,],store[^fetch),
R3. (right,xy,fetch) ⇒ (left,y,store x), x,y ∈ {o,]},
R4. (right,xy,store z) ⇒ (left,zxy,fetch), xy ≠ [2, z≠[,
R5. (right,xy,store [) ⇒ (left,[xy,ε), xy ≠ [2,
R6. (right,ox,store]^fetch) ⇒ (left,x,store]^store o), all x,
R7. (right,$]_1$,store]^fetch) ⇒ (left,$]_1$]2,fetch fetch),
R8. (right,x,store[^store o) ⇒ (left,ox,store[^fetch), x ≠ [.

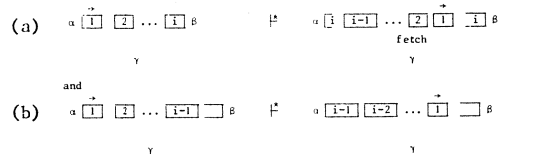
L1. (left,2],top) ⇒ (right,2,top store]),
L2. (left,][,fetch) ⇒ (right,[,store]^fetch),
L3. (left,xy,fetch) ⇒ (right,x,store y), x,y ∈ {o,[},
L4. (left,xy,store z) ⇒ (right,xyz,fetch), xy ≠ 2], z ≠],
L5. (left,xy,store]) ⇒ (right,xy],ε), xy ≠ 2],
L6. (left,xo,store[^fetch) ⇒ (right,x,store[^store o), all x,
L7. (left,$[_1$,store$[_2$^fetch) ⇒ (right,$[_2[_1$,fetch fetch),
L8. (left,x,store]^store o) ⇒ (right,xo,store]^fetch), x ≠].

We choose the rules so that they gave a concise description. Situations not covered by the rules do not occur in the operation of the device; certain situations covered by the rules do also not occur; all situations covered are covered uniquely: only one rule is applicable. The rules were also checked by having a simulation on 9 blocks of size 3, i.e. '[o]', run for > 10^5 steps, going 8 levels deep in recursion, i.e. up to starting to transport cells of block 9. Assuming that M maintains a variable 'lastmove' in the finite control, at each step M performs:

Algorithm MOVE:

select appropriate transition rule and execute it.

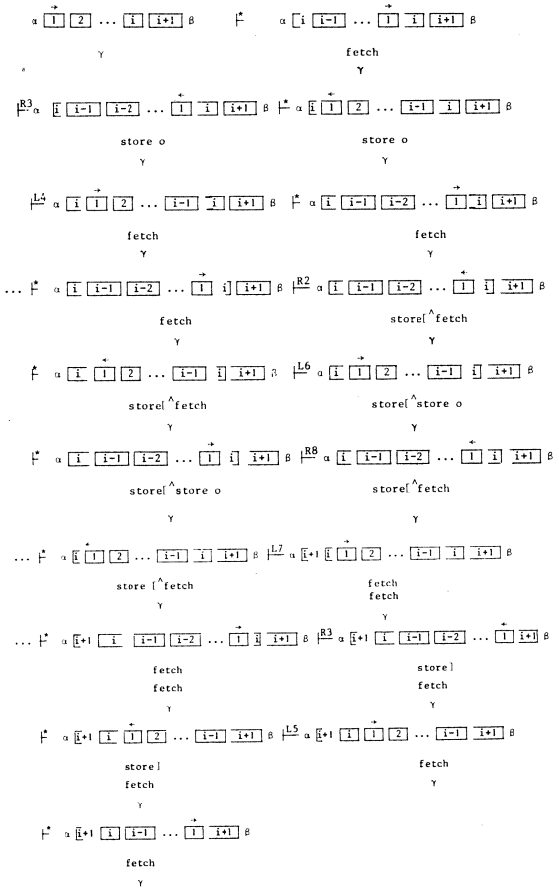
Lemma 1. *For all* i > 1, *and all* α,β,γ *where* α *is the tape left of super block I,* β *the tape right of super block* I *and* γ *the stack contents of the displayed start instantaneous descriptions:*



Similarly from right to left, i.e., the mirror image of the depicted situations.

Proof sketch. *Base case*: i = 1 and i = 2, we leave for the reader to verify using the given rules.

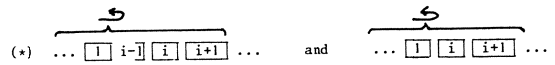
Induction: Assume that the Proposition holds for all j ≤ i. Then:



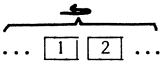
So (a) holds for j = i+1.
Similarly we prove (b), and so the Lemma. □

Using Lemma 1, we can prove by induction on the number of steps:

Lemma 2. *For all* i ≥ 1, *superblock* I *always is a connected tape segment which is continuously traversed from the left end to the right end and back to the left end. Each such full sweep takes* S(i) ≤ 24^i *steps. For* i > 2, *during such a full sweep over superblock* I+2 *the subconfigurations*

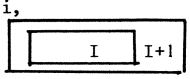


both occur. For i = 2 (*) *right occurs during a sweep over superblock* 4, *and for* i = 1 *the sub-configuration*



occurs during a full sweep over superblock 3. *Since* ℓ = 25 *superblock* 2 *is always completely covered. Each pair of adjacent closed blocks right of block* 1 *are blocks* i *and* i+1 *in that order.*

So the described one-head tape unit with square insertion/deletion and with the attached pushdown store can support an abstract storage unit S satisfying the Requirement.
To allow a more intuitive understanding of the process, note that there is a definite topology associated with the tape configurations: always, for all i,



Here (I+1) - I = the cells of block i+1 present on the tape, in the correct left to right order.
The order of blocks is maintained by being aware where block 2 is, and by linking blocks i and i+1 via ']['adjacent borders and linking the associated instructions on the stack.

Eliminating the square insertion/deletion facility and the pushdown store

We accomodate the square insertion/deletion facility by absorbing deleted squares in block 1 and extracting squares to be inserted from block 1. The deleted squares are kept as a stack in the different blocks, block 1 containing at most 3 and each consecutive block at most 2. The regular block traversal allows us to pass deleted squares from block to block, so as to have the stack of deleted squares with the top in block 1 grow and shrink as required. (The effect of a square deleted/inserted in block i+1 does not affect the amount of deleted squares in blocks j, j ≥ i+1.) Similarly, and with somewhat less fluctuations, we can maintain the pushdown store contents as a stack on blocks 0,1,..., of which the top is contained in block 0. The information on the pushdown store is attached to the resident cells of the blocks. Let the *occupancy* O(i) of a block i be the number of *free* (i.e. deleted) squares it stores. Then the procedure outlined below will ensure that $0 \le O(1) \le 3$ and $0 \le O(i) \le 2$ for all i > 1 at all times. Squares which are deleted on the tapes are absorbed by block 1 and squares to be inserted are released by block 1. Transport of free squares from block to block, so as to have the stack of free squares with top in block 1 grow and shrink as required, is effected by the head whenever it scanns two adjacent closed blocks. As previously argued, the process maintains the property that all adjacent closed blocks i,j right of block 1 are in the correct left-to-right order: j = i+1, and all adjacent closed blocks, i,j left of block 1 are in the correct right-to-left order: j = i-1. Block 1 always knows which of an adjacent closed block (if there are two of them) is block 2. Whenever the head on block 1 is in an end position, and it scanns 2 adjacent closed blocks on that side, say block i and i+1, it first distributes free squares as follows:

$O(i) > 1 \Rightarrow O(i) \leftarrow O(i)-1 \;\&\; O(i+1) \leftarrow O(i+1) + 1$

$O(i) = 0 \;\&\; O(i+1) > 0 \Rightarrow O(i) \leftarrow 1 \;\&\; O(i+1) \leftarrow O(i+1)-1$

Whenever block 2 is closed, the free squares in blocks 1 and 2 are immediately adjusted according to the above distribution scheme, when block 1 absorbs or exudes a free square. Clearly the distribution of free squares keeps the total number of free squares on the tape correct.
We define a *chain* on the tape as a sequence of closed blocks $i_1, i_2, \ldots, i_k$ on the tape with i_j adjacent to i_{j+1}, $1 \le j < k$. If i_1 is the one nearest block 1 (or i_1=1) and i_k is the one furthest from block 1, then $i_{j+1} = i_j+1$ since every pair of adjacent closed blocks have such indexes. If $i_1 = 1$ the chain is *grounded*, and if a chain is not a proper subchain from any other chain on the tape it is called maximal. We denote a maximal grounded chain as a MGC. We define 2 invariants (4a&b) and (5).

(4a) Every MGC C of length i contains at most i+2 free squares. If there are $s \le i$ free squares on the total tape C contains all of them. More in particular:
- (i) top pd store = fetch ⇒ C contains min (i,s) free squares
- (ii) top pd store = store. ⇒ C contains min (i+1,s) free squares
- (iii) top pd store = store.^fetch ⇒ C contains min (i+1,s) free squares
- (iv) top pd store = store.^store. ⇒ C contains min (i+2,s) free squares.

For any MGC C there is a time *before transaction* with environment and a time *after transaction* with environment. The time before transaction is the time when the head on block 1 is proceeding to the border of block 1 which is not adjacent to an element of C. If C has length 1 both borders qualify as such. We call C at the time before transaction C_b.

(4b) The free squares are distributed over the blocks in C_b as follows:
$s \le i$: one free square per block from block 1 up to block s and no free square in blocks s+1 up to block i.
$s > i$: (leftmost digit O(1) and rightmost O(i).)
- (i) top pd store = fetch: 11...11
- (ii) top pd store = store.: 11...12
- (iii) top pd store = store.^fetch: 11...12
- (iv) top pd store = store.^store: 11...12 (s=i+1); 11..122 (s≥i+2)

this in case i > 1. If C has length 1 then all free squares in C are of course contained in block 1.

(5) For all i > 1, $0 \le O(i) \le 2$, and $0 \le O(1) \le 3$.

The consequence of the maintenance of invariant (4) is that at any time the head on block 1 needs to insert a square, block 1 contains at least 1 free square. Also, whenever the head on block 1 has to delete a square, block 1 can absorb a square without violating (5). The fact that invariants (4) and (5) are maintained will show that the traversal of the blocks allows us to keep a stack of free squares on the consecutive blocks with the top in block 1, so that the square deletion/insertion facility is incorporated in our ordinary tape, without needing

to expand block 1 by more than 3 squares and the other blocks by more than 2 squares.

Lemma 3. *Invariants* (4) & (5) *are maintained by the 'free square' distribution rules.*

Lemma 3 is proved by induction on the number of steps and shows we can maintain a stack of free squares on the consecutive blocks, while not expanding any block over the allowed limit.
Similarly, and actually by the same distribution rules, we maintain the extra pushdown store as a pushdown store on the consecutive blocks by attaching its entries to the resident cells of the blocks. An element is added to the pushdown store when either block 2 is opened or when a 'store.^fetch' is replaced by a 'fetch fetch'. An element is popped from the pushdown store when a block is closed, i.e., when 'store]' or 'store[' is replaced by ε. In this case we also use block 0, to contain the top of the pushdown store, so here the occupancy of all blocks $i \geq 0$ needs only be at most 2.
Thus we have eliminated the square deletion/insertion facility and the extra pushdown store by incorporating them in the one-head tape unit.

Eliminating the fat head. By using Hartmanis-Stearn's construction [3] of cutting out $2\ell+1$ squares covered by the head, maintaining them in the finite control and exchanging contents with the single square covered on the tape as required, we eliminate the assumption of the fat head covering $2\ell+1$ squares. Note that, for the various update /distribution strategies discussed, we never have to look farther ahead then over the first element adjacent to block 1, and cover the next adjacent 2 blocks completely. So $\ell = 25$ suffices.
By Lemmas 2 and 3 the following one-head tape unit supports an abstract storage unit S meeting the Requirement. Let M be as described and perform at each step:

Algorithm STEP:

step 1 check for unmarked squares to create new blocks of cells
step 2 distribute free squares
step 3 distribute elements of the pushdown store
step 4 MOVE
(The first execution of STEP preceded by initialize block 0 and block 1 and put 'fetch' on the bottom of the simulated pushdown store.)

Proposition 4. *The constructed one-head tape unit* M *supports an abstract storage unit* S *meeting the Requirement.*

Since M has as yet no input-output connections, its head movement is a function of time alone and therefore *oblivious*. Equipping M with input/output terminals, and plugging in an appropriate piece of finite control for each multicounter machine to be simulated, having the resulting machine execute the following algorithm:

INITIALIZE; REPEAT: (read input; UPDATE counters; write output; STEP)

we have by Propositions 1-4:

Theorem. *For each* $k \geq 1$, *each* k*-counter machine can be simulated in simultaneous real-time and logarithmic space by an oblivious one-head tape unit.*

3. Conclusion.

We have made no attempt to obtain optimal block sizes and amount of squares covered by the head, but instead aimed at a palatable exposition of the result. We now place it in context.
For various theoretical and practical reasons, multitape Turing machines, restricted in one or more resources, serve as a standard against which to calibrate the power of other devices or to compare power amongst themselves under different resource restrictions. The commonly considered resources are time, space, number of tapes, number of head reversals and oblivious-nonoblivious. The present simulation is unusual in that it is optimal in all of these resources; the use of no resource can be improved by relaxing on the other resource restrictions. Because of the many applications of multicounters we can expect the result to yield additional fruit. For instance, we now can in real-time keep track of which pairs, out of a collection of k, $k > 1$, stacks, are of equal heigth using one extra (oblivious) one-head tape unit or two extra (oblivious) pushdown stores. This task formerly required k-1 extra nonoblivious pushdown stores (and is impossible for a single pushdown
store receiving only information concerning the stack movements). Similar statements can be made about the head positions in multitape Turing machines.

Number representations. The result yields an unusual number representation. It shares compactness of notation with positional number systems and ease of incrementation/decrementation by 1 and instant zero check with tally systems and variants. It improves both in that many integers can be maintained under instant incrementation/decrementation by 1 and instant zero check, using a single access pointer in the linear notation which never moves more than $\lceil \log n \rceil$ positions in any segment of time of less than n units. Moreover, the movement of the pointer is independent of the input, and $((3k+O(1))\log n$ bits are used to denote k n-bit integers.
Under an actual implementation of a multicounter machine therefore the time cost is proportional to the number of counters maintained and not to the size of the counts.

References

[1] M.J. Fischer & A.L. Rosenberg, *Real-time solutions of the origin-crossing problem*, Math. Systems Theory 2 (1968), 257-264.

[2] P.C. Fischer, A.R. Meyer & A.L. Rosenberg, *Counter machines and counter languages*, Math. Systems Theory 2 (1968), 265-283.

[3] J. Hartmanis & R.E. Stearns, *On the computational complexity of algorithms*, Trans. Amer. Math. Soc. 117 (1965), 285-306.

[4] M. Minsky, *Recursive unsolvability of Post's problem of tag and other topics in the theory of Turing machines*, Ann. of Math. 74 (1961), 437-455.

[5] N. Pippenger & M.J. Fischer, *Relations among complexity measures*, Journal ACM 26 (1979), 361-384.

[6] C.P. Schnorr, *The network complexity and Turing machine complexity of finite functions*, Acta Informatica 7 (1976), 95-107.

[7] P.M.B. Vitányi, *Efficient implementation of multicounter machines on oblivious Turing machines, acyclic logic networks, and VLSI*, Techn. Report IW 167, Mathematisch Centrum, Amsterdam, May 1981. Also: P.M.B. Vitányi *Efficient simulations of multicounter machines*, Proc. 9-th ICALP, Lecture Notes in Computer Science, Springer Verlag, 1982.