Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

L.M. Kirousis, E. Kranakis, P.M.B. Vitanyi

Atomic multireader register

Computer Science/Department of Algorithmics & Architecture Report CS-R8722 May

69B31, 69B43, 69D51, 69D52, 69D54

Atomic Multireader Register

Lefteris M. Kirousis

University of Patras, Department of Mathematics, Patras, Greece
and
Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands
(lefteris@cwi.nl)

Evangelos Kranakis

Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands
(eva@cwi.nl)

Paul M.B. Vitányi

Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands
(paulv@cwi.nl)

ABSTRACT

We give implementations for atomic, shared, asynchronous, wait-free registers: (i) A new implementation of an atomic, 1-writer, 1-reader, b-bit register from $O(b)$ safe, boolean registers (i.e., from scratch). The solution uses neither repeated writing of the input nor repeated reading of the output. (ii) An implementation of an atomic, 1-writer, n-reader, multibit register from $O(n^2)$ atomic, 1-writer, 1-reader, multibit registers. Both constructions rely on the same idea. In a sense (ii) is a generalization of (i). These results show how to construct atomic, multireader registers from - basically - elementary hardware like flip-flops.

1980 Mathematics Subject Classification (1985 Revision), Primary: 68M10, *Secondary: 68P15, 68Q25.*
CR Classification System (1987 Version): B.3.2, B.4.3, D.4.1, D.4.2, D.4.4.
Keywords and Phrases: Register, shared register, atomic, regular, reader, writer.
Note: This paper will be submitted for publication elsewhere.

1. Introduction.

We are interested in true concurrency in the context of shared register access by asynchronous processors. Concurrency control of asynchronous processes is often realized by actively serializing concurrent actions, using synchronization primitives like mutual exclusion, semaphores, and locking. Thus, although it *seems* that the actions are executed concurrently, deep in the system

The work of the third author was supported in part by the Office of Naval Research under Contract N00014-85-K-0168, by the Office of Army Research under Contract DAAG29-84-K-0058, by the National Science Foundation under Grant DCR-83-02391, and by the Defence Advanced Research Projects Agency (DARPA) under Contract N00014-83-K-0125.

Report CS-R8722
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

they are *actually* executed serially in some order. It has been pointed out by Lamport (1986) that to implement such primitives, we first need interprocess communication through a shared memory register, even if the processors communicate by message passing. This suggests that the problem of simultaneous memory access needs to be solved without recourse to synchronization primitives. It is desired that such a solution involves *no waiting* by one operator for another one. Thus we kill two birds with one stone, since it is the waiting involved in synchronization primitives which may make solutions using them unacceptable. For instance, in case a fast computer (like a Cray) communicates with a slow computer (like a PC), using a readers-writers protocol, forced waiting may slow down the Cray several orders of magnitude.

The problem of providing general, wait-free, asynchronous communication interfaces becomes more acute, as more and more hardware from different technologies, scale and speed continue to be connected in computer networks and other complexes. Note that asynchrony need not be due solely to hardware, but can also be caused by multiple users on the various machines.

In this paper, we solve the problem of how to implement a shared register which can be read by different asynchronous processors (the readers) and be written by one asynchronous processor (the writer) in a truly concurrent fashion. That is, without any restrictions to prevent simultaneous access and making no assumptions about the relative durations of the readings and writings. Apart from intuitively explaining our algorithms, we provide *rigorous* and complete proofs of their correctness.

1.1. The Model.

In this section, we informally describe the model we use. For a formal development, the reader is referred to Lamport (1986) (or Awerbuch et al (1987)).

A **register** is a storage element, where a number of processors can either read or write. A **writing action** writes (or stores) onto the register a value from a given domain, while a **reading action** reads (or returns) a value from the same domain. Thus, with every action we associate a value from the domain of the register.

In this paper, we study registers where n processors, the **readers**, can read, while only one processor, **the writer**, can write. The writer is different from the readers. Thus, altogether, we have $n+1$ processors. Notice that we can assume that the writer can read as well. This is so, because the writer can store the value it writes into its local memory, and so at any instant can retrieve the last value it wrote. This is a valid value to read, since it is only that single writer who can change the contents of the register. This remark would not be true if we had more than one writers.

The notion of the size of the domain of the register is captured by the number of bits that the register can hold. For notational convenience, we call a register with 1 writer, n readers and b bits a $1\mathrm{W}n\mathrm{R}b\mathrm{B}$ register. We do not necessarily assume that in a b-bit register a value from the domain is written on b separate bits. The number of bits determines only the size of the domain and is not related to the actual architecture of the register, which could be quite complicated.

A **sequential** register is one where all actions are *a priori* required to be executed sequentially. In a sequential register, a reading r returns the value that was written onto the register by the last writing preceding r.

We are mainly interested in non-sequential registers, i.e. registers where actions by different processors can be concurrent. However, actions by the same processor are again assumed to be executed in a sequential fashion. To implement a non-sequential register, since we may have concurrent actions, some means of communication between the processors becomes necessary. In our model, this communication is attained by having the processors write or read messages onto **subregisters** of the register. This exchange of messages is carried out following a **protocol.**

To distinguish a register from its subregisters, we call the former a **compound** register. Also the actions on the subregisters are called **low-level** actions (or **subactions**), as opposed to the **high-level** actions on the compound register. The subregisters are themselves registers, but in general, have weaker properties than the compound register.

We do *not* assume that there is a location where at a given instant the current value of the compound register appears. Actually, we assume that the compound register consists only of the subregisters where the processors communicate. We further assume that a high-level action consists only of a series of low-level message readings and writings on the subregisters (a high-level reading, and a high-level writing too, comprises, in general, both subreadings and subwritings). Thus, the compound register is an abstract or conceptual register. It is the protocol that relates the value associated with a high-level action with the values of its constituent subactions. The subregisters of the compound register may themselves be non-sequential. Therefore, they may consist of subregisters at an even lower level. Nevertheless, when working with a compound register, we do not take into account the specific architecture of its subregisters. We consider them as conceptual registers with the properties they are assumed to have.

For notational convenience only, we refer to the elements of the domain of the compound register as **words.** Also, while we use the expression 'reads a value' to describe both a high-level and low-level reading, we reserve the expression 'returns a word' only for high-level readings. This is because low-level readings are viewed as simple message readings, while high-level readings are viewed as output returnings. Formally, there is no difference between the two.

We assume complete asynchronicity among different processors. The subactions of a particular high-level action must be executed sequentially in the order specified by the protocol. But, apart from that, they are 'free' to take place at any time 'they choose'. That also means that there is no waiting of an action (high- or low-level) for another. Thus, subactions from different high-level actions may interleave or overlap at an arbitrary fashion. Notice that since the subregisters are not assumed, in general, to be sequential, we may have concurrent subactions onto the same subregister (by different processors).

With every action we associate a non-empty time-interval (i.e., an interval of positive real numbers), during which the action is assumed to take place. This interval is called the **duration** of the respective action. The duration of an action can even be a singleton. In that case, the action is called **instantaneous.** An action a is said to **precede** an action b (notationally, $a \rightarrow b$) if every instant (real number) in the duration of a is strictly less than every instant in the duration of b. The actions a and b are said to be **concurrent**, if their durations are not disjoint. In a compound register, the duration of a high-level action a is by definition the smallest interval that covers the durations of all subactions of a. The relation $\rightarrow$ is a strict partial ordering. For the 1-writer registers we study, it *linearly* orders the set of writings. This is so, because actions by the same processor are executed sequentially.

Lamport (1986) defines the precedence relation among actions in an axiomatic way, without any reference to time. Actually, his precedence relation is causal rather than temporal. This is justified by the fact that parts of distributed systems may be located far apart and that signals travel with a speed close to the speed of light. Because in this paper we are mainly interested in the algorithmics of various protocols, we choose to work with a temporal precedence relation.

Thus, proofs of correctness are considerably simplified. Notice that we do *not* assume that the processors have access to a global clock. Time is exclusively used to define the notions involved and to prove the correctness of the algorithms. Awerbuch et al (1987) give certain correctness proofs following essentially Lamport's definition of causal precedence relation. The interested reader can, from that paper, see how the transition from temporal to causal notions is carried out.

An arbitrary protocol on a compound register may associate with a high-level reading a value outside the domain of the register or an outdated value. Even worse, we may have global inconsistencies in the way values are assigned to readings, e.g., we may have two readings (the first preceding the second) such that the first returns a value more recent than the value returned by the second. This may happen when both readings are concurrent with the writing that writes the recent value. Thus, the first reading may pick the recent value, while the second may choose the previous one. This phenomenon is called a new-old inversion. Following Lamport (1986), we classify the registers and their protocols into categories that reflect how consistent are the values associated with the readings.

A **run** (or history) ρ of a register is a sequence of actions executed according to the protocol. We denote by R the set of readings of ρ and by W its set of writings. We allow different writings of a run to write equal values. We assume that for any run ρ and for any action a of ρ, the set $\{b: \text{not}(a \rightarrow b)\}$ of actions of ρ is finite. Therefore, since $\rightarrow$ linearly orders the writings, every writing w has a unique writing that directly precedes w, and a unique one that directly follows it (unless w is the first or the last writing, respectively). These are denoted by w^- and w^+, respectively. Let now r be a reading and w a writing in a run ρ. We also say that w **directly precedes** r if a) $w \rightarrow r$ and b) there is no other writing w' so that $w \rightarrow w' \rightarrow r$. Since $\rightarrow$ is a total ordering on the set of writings, there is no more than one writing directly preceding a reading.

A run ρ is called **safe** if every reading r of ρ that has no concurrent writings returns a value which is equal to the value written by a writing that directly precedes r.

A safe register is one with minimum requirements. In case of no concurrency of a reading with writings, the reading returns a recent value. Notice that safety does not imply anything about readings that are concurrent with writings.

A run ρ is **regular** if every reading r returns a value which is equal to the value written by a writing that is either concurrent with r or directly precedes r.

In a regular run, all readings return recent values. But still, we may have new-old inversions.

A run is **atomic** if there is a total ordering $\Rightarrow$ such that

a. $\Rightarrow$ extends the precedence relation $\rightarrow$ defined on the actions of the run (**external consistency**) and

b. every reading r of ρ returns a value which is equal to the value written by a writing that directly precedes r in the sense of the total ordering $\Rightarrow$ (**internal consistency**).

A register is called atomic (respectively, regular, safe), if every run that follows the protocol is atomic (respectively, regular, safe).

Atomicity is the strongest possible condition. It guarantees that the actions take place *as if* they were executed sequentially. Obviously, new-old inversions are excluded from an atomic run. Notice that an atomic run is regular and a regular run is safe. Moreover, as is proved in Awerbuch et al (1987), in an atomic run, for any action a we can define an instant (real number)

$\sigma(a)$, belonging to its duration, such that a can be considered to take place instantaneously at $\sigma(a)$. Formally, that means that if we order all actions according to their σ-value, then any reading reads the value of the writing that directly precedes it in that order. A similar formalization of the notion of atomicity is given by Misra (1986). The notion of register atomicity is closely related to what is called '(strict) serializablity' in concurrency control, in particular in the context of databases with concurrent 'transactions' (see, e.g., Papadimitriou (1979)).

In the definitions above, the readings were associated with a writing that writes a value equal to the value returned by the reading. Formally, a **reading function** for a run ρ is a partial function $\pi\colon R \to W$ such that if $\pi(r)$ is defined, then $\pi(r)$ writes a value equal to the value that r returns. We say, by a slight abuse of terminology, that r reads $\pi(r)$.

Obviously now, a run is safe if there is a reading function such that for every r that has no concurrent writings, $\pi(r)$ is defined and directly precedes r. A run is regular, if there is a total reading function π such that for every r, $\pi(r)$ is either concurrent with r or directly precedes it. Finally, a run is atomic, if besides a total ordering $\Rightarrow$ that extends the precedence relation $\to$, there is also a total reading function π that respects the total ordering $\Rightarrow$ (in other words, $\pi(r)$ directly precedes r in the total ordering).

For 1-writer registers, we have a much more manageable atomicity criterion than the one of the above definition. This **atomicity criterion**, due to Lamport (1986), states that a run is atomic if there is a total reading function π such that the following three conditions are satisfied:

F. For any reading r, not $(r \to \pi(r))$.

P. For any reading r, there is no writing w such that $\pi(r) \to w \to r$.

I. If r_1 and r_2 are two readings such that $r_1 \to r_2$, then not $(\pi(r_2) \to \pi(r_1))$.

Condition (F) states that there is no reading in the Future. Condition (P) (which, together with (F), is the regularity condition) guarantees that there is no reading in the overwritten Past. Finally, condition (I) states that we cannot have new-old Inversions. Intuitively, it is easy to see that once these conditions are satisfied, then the register is atomic. Indeed, we can place all readings immediately after the writing they read (the writings are totally ordered). The readings that are thus put immediately after the same writing can then be ordered by arbitrarily extending the partial precedence ordering among them. Nevertheless, the reader who would prefer to see a formalization may consult Lamport (1986) (or Awerbuch et al (1987)). It is the above three conditions that we will prove, whenever we would like to show that a register is atomic.

1.2. History of the Problem and Results of the Paper.

The main objective in the area is to implement registers with strong properties, like atomicity and accessibility by many processors, from registers with weaker properties. The simplest register is the safe, 1-writer, 1-reader, boolean (1-bit) register. This register models the most elementary hardware, i.e., a flip-flop. The safety condition guarantees that in our model, the behavior of the building units under simultaneous access is irrelevant for the final output †.

† Let us stress here that safe, 1W1R1B registers are mathematical concepts, while flip-flops are physical objects. Bistable devices like flip-flops are prone to *metastable* operation, see e.g., Marino (1981) or Chapiro (1984). If the input that causes the bistable to change state is marginal, it may leave the bistable in a metastable state (or metastable region) different from the two stable states. There it may remain for an indefinite time, before resolving to one of the stable states. We do not consider this level of physical detail here, but consider idealized bistables.

Lamport (1986) implemented an atomic, 1W1RbB register from safe, 1W1R1B registers. For his construction he used Peterson's (1983) implementation of an atomic, 1WnRbB register by a) 2 safe 1WnRbB registers, b) n safe 1W1RbB registers c) 2n atomic 1W1R1B registers and d) 2 atomic 1WnR1B registers. Essentially, Lamport constructs atomic, 1W1R1B registers from safe ones, and then applies Peterson's result for $n = 1$.

The case of registers with more than one writers is studied by Bloom (June 1986), Vitányi and Awerbuch (October 1986), Peterson and Burns (December 1986) and Awerbuch et al (February 1987). However, the problem of implementing an atomic, 1-writer, *multireader* register from atomic, 1-writer, 1-reader registers was unresolved. We solve this problem in Section 3. After our result appeared in Kirousis et al (January 1987) an independent solution of the same problem by Singh et al (December 1986) came to our attention. Their method is completely different. An advantage of our implementation is that the main new idea involved gives also a novel and simple implementation of an atomic 1W1RbB register from safe 1W1R1B registers. The latter construction, which we believe is significant in its own right, is presented first in Section 2. It has the advantage that it does not require repeated writing of the input nor repeated reading of the output, like previous constructions to the same effect do. The importance of this stems from the fact that it pays to use such constructions mainly for a long input (and output). Presenting this construction first helps to make clearer the intuition behind our construction of an atomic, multireader register.

2. The 4-Tracks Protocol.

In this section, we show how to implement an atomic 1W1RbB register from safe 1W1R1B registers. Since we are restricted to use only one-bit registers, the writer, in order to write a b-bit word, must use at least an array of b 1-bit subregisters. We call such an array a **track**. The track-subregisters are written by the writer and read by the reader. Observe that if a reader starts reading the contents of the subregisters of a track, while the writer has not yet written all of them, a value outside the domain of the compound register may be returned. This is so, because the reader may read some subregisters of the track which have not yet been changed by the writer and some which have been. To avoid this situation, our protocol guarantees that while the writer is writing on a track, the reader does not do anything there. But then, since there should be no waiting, there must be more than one track. Thus, if a track is occupied by one processor, the second processor is referred to another track. In our construction we use four tracks (numbered 1 through 4). Each high-level action has subactions taking place on a single track. We call **track-duration** of a high-level action a the smallest interval that contains the durations of all subactions of a that are executed on a track-subregister. Our protocol then guarantees the following:

CF. The track-durations of any two high-level actions whose subactions are executed on the same track are disjoint.

The above intuitively states that working on a track is Collision-Free. As we will see, once (CF) is satisfied, we can define a reading function π. Then, we have to make sure that conditions (F), (P) and (I) of the atomicity criterion are satisfied. To accomplish these objectives, the processors decide which track to go to by reading and writing on certain subregisters that comprise what we call a **switch**. The switch consists of an array (RQ_l, A_l), $l = 0,..., k$ of pairs of subregisters RQ_l and A_l (Figure 1). The subregisters A_l are written by the writer and read by the reader. On the contrary, the subregisters RQ_l are read by the writer and written by the reader.

Roughly, the writer in order to write a word, consults the switch and decides which track to

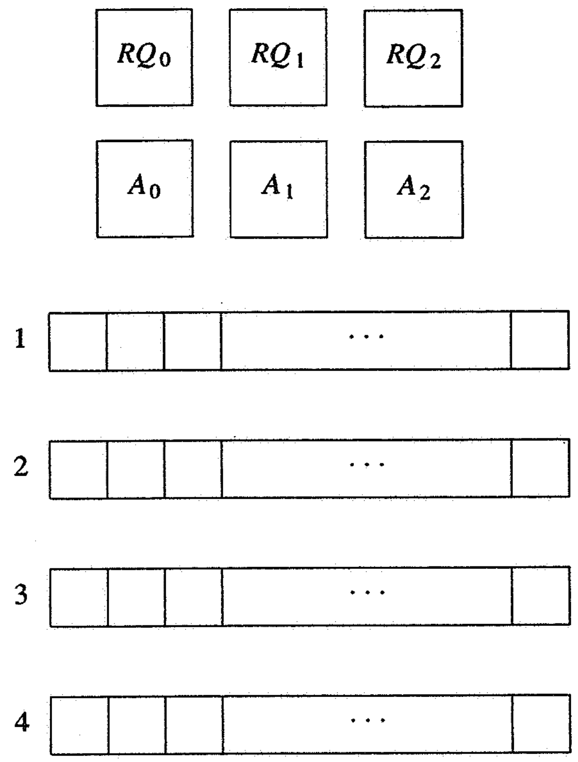

Figure 1: The 4-tracks register.

go to. Then it writes the bits of the word on the subregisters of this track. Finally, it writes on a subregister of the switch the number of the track it used. On the other hand, the reader first does certain subreadings and subwritings on the switch in order to decide which track to go to and to inform the writer of its intention to read. Subsequently, it reads a word by reading the bits of the track it has decided to go to.

We refer to the subregisters RQ_l and A_l as the subregisters at **layer** l. The subregisters of the tracks are right from the beginning assumed to be safe 1W1R1B subregisters. But to make clear the ideas involved in the construction of the switch, we initially assume that some of its subregisters are atomic 1W1R registers with a constant (independent of b) number of bits. Subsequently, we show that we may assume that they are only regular. So, by a construction of Lamport (1986), they can be implemented by safe 1W1R1B registers. Also, again in order to make the presentation clearer, we initially assume that we have an unbounded number of layers. This, as we show, creates no problems, because both processors move within a bounded 'window' of layers. So, layers outside the scope of this window can be recycled, and therefore, a bounded number of them (actually, only three) suffices.

2.1. The Algorithm for a Switch with Unboundedly Many Layers and Atomic Subregisters.

In this section we assume that we have unboundedly many layers (numbered 0, 1,...). We assume that the subregisters A_l are atomic 1-writer, 1-reader registers which can hold a value from a set of five élements. The RQ_l's are assumed to be regular, one-bit registers, while the track-subregisters are assumed to be safe boolean registers. The five possible values of the A_l's are E

(for **empty**), 1, 2, 3 and 4. The value E signifies a subregister where the writer has not written yet; the other four signify track numbers. The writer, at each high-level writing, writes on a certain A_l the number of the track that it visited. The two possible values of the RQ_l's are P (for Please) and E. Again, E signifies an empty subregister not yet used by the reader. On the other hand, P signifies a message to the writer to write its track-number to the subregister A_l of the next layer. The protocol guarantees that at any instant, the switch-subregisters that were accessed last by the two processors are at identical or adjacent layers.

The switch-subregisters are all initialized by the value E. The track-subregisters are arbitrarily initialized. We assume that the initial value of any subregister used in a run is written by an initializing subaction that precedes all other subactions on this subregister (thus the reading function on the switch-registers is always defined). An initializing subaction is not part of a high-level action. We also assume that there is a high-level writing by the writer that precedes all other high-level actions. We do not consider it as an initializing action.

Each processor has certain **local** variables to which only itself has access. The protocol gives not only the sequence of subactions of each processor on the subregisters (called also **shared variables**) but also the sequence of actions on the local variables. For notational uniformity, we call these 'subactions', as well. Notice though, that the subactions on local variables are not taken into account for the definition of the precedence relation among high-level actions or the definition of the notion of concurrency. This is so, because the subactions of a processor on its local variables do not directly influence any of the subactions of the other processor. Nevertheless, high-level actions by the same processor not only are not concurrent, but also, their subactions on local variables do not interleave.

The writer has a local variable wl which signifies the layer it is going to use. It is initialized by the value 0. Similarly, the reader has a local variable rl, which signifies the layer that the reader is going to use. It is initialized by the value -1 (the first subaction of the reader is $rl := rl+1$). The writer has two local variables wt and wt^-. The first denotes the track-number that the writer used last. The second denotes the track-number used at the high-level writing preceding the last. Both wt and wt^- are assumed to have no value initially. The reader has a local variable rt that signifies the track-number it is going to use. It is assumed to have no value initially. Finally, the writer has a local variable vb (verboden) which is a set containing at most two track-numbers where the writer is forbidden to go. Initially, vb is assumed to be the empty set. The protocol is given in Figure 2.

2.2. Proof of Correctness.

As stated in the introduction, each subaction is assumed to have a duration. Since later we are going to show that subregisters A_l can be assumed to be only regular, we refrain from considering the actions on them as instantaneous, although, in this section, these subregisters are considered to be atomic. So, we may have concurrent *subactions* on the same subregister A_l. We will use the atomicity assumption only in order to show that no new-old inversions happen and we are going to explicitly mention where we use it. The beginning (respectively, the ending) instant of a subaction is the left (respectively, right) end-point of its duration. Subactions on local variables are assumed to be instantaneous. This is no loss of generality, since they are executed sequentially by the respective processors.

We say that the reader executes a **backup**, when it sets $rl := rl-1$. Similarly, we say that a processor executes an **advance**, when it increments by one its layer variable. The writer never

```
The writer writes a word $v$:
1.  read $RQ_{wl}$;
2.  if $RQ_{wl} = P$ then do $wl := wl+1$; $vb := \{wt, wt^-\}$ od; /*if the writer reads $P$ on $RQ_{wl}$—an
    indication that the reader could only be at a track whose number appeared on $A_{wl}$—it moves
    one layer ahead and it stores into the forbidden set of track-numbers the two track-numbers
    it visited last—these are the tracks where the reader could be */
3.  $wt^- := wt$; $wt := \min(\{1,2,3,4\} - (\{wt^-\} \cup vb))$; /*the writer chooses a non-forbidden track-
    number, which is also different from the one it visited last*/
4.  write the bits of $v$ onto the track with number $wt$;
5.  write $wt$ on $A_{wl}$ /*the writer forwards to the reader the track-number it just used*/.

The reader returns a word:
1.  $rl := rl+1$; /*the reader starts by advancing to a new layer*/
2.  read $A_{rl}$;
3.  if $A_{rl} = E$ then do
            $rl := rl-1$; /*if the reader finds its new layer empty, it backs up to the previous one*/
            read $A_{rl}$ and store its value into $rt$ od
    else do
            write $P$ on $RQ_{rl}$; /*$P$ is a message to the writer to advance to a new layer, thus the
            reader can, 'at its leisure', re-read $A_{rl}$. The only writing that can change $A_{rl}$ from now
            on is one which is not yet completed and had checked $RQ_{rl}$ before the printing of $P$ */
            read $A_{rl}$ and store its value into $rt$ od; /*the reader, after it has written the message $P$,
            reads a track-number from $A_{rl}$ */
4.  read the bits of track with number $rt$ and return the word thus obtained.
```

Figure 2: The protocol for the two processors.

backs up. Observe that after the reader advances to a layer l, it checks A_l. If it finds it empty, it backs up to the previous layer. Therefore, if the reader ever executes an advance *from* a layer l, it must have read a nonempty value at the last execution of step (2) of its protocol on l. Therefore, we get the following:

Lemma 2.1. The reader, during an execution of step (3) of the reader's protocol, stores into rt a value $\neq E$.

Proof. The lemma is obvious, when there is no backup. Notice however that we are using the fact that no new-old inversions can occur on A_l (A_l is atomic). On the other hand, if a backup to A_l takes place, then since the reader has previously advanced from A_l, we have from the previous remarks, that a nonempty value has been previously read from A_l. So, again using the atomicity of A_l, we have that at step (3), even with a backup, a value $\neq E$ is stored.□

Notice that in the above lemma we use, for the first time, the atomicity assumption of the subregisters A_l. By a similar reasoning (without using any atomicity assumption), when the reader advances to a layer l, it must have completed the writing of a P at the previous layer $l-1$. Observe also that on each RQ_l there is a unique writing of a P. Similarly (again without using any atomicity), if the writer executes an advance *from* a layer l, it must have previously read a P on RQ_l. We state now (and prove at the end of the section) the following:

Lemma 2.2. Condition (CF) is satisfied.

According to the protocol, the reader is always instructed to go to a track where the writer has *previously* written. This is so, because the writer prints its track-number *after* it has finished writing its word there. But, *prima facie*, a fast writer may return to this track, while the slow reader is still there. This is not the case though. Indeed, informally, if during a high-level writing w the writer sees a P, it moves to the next layer. Thus, the reader that wrote that P can only go to a track whose number was written by a writing w' preceding w. As we will prove, w' can only be w^- or w^{--}. But the writer stores the track-numbers of w^- and w^{--} into vb and thus it avoids them. So, (CF) is true. We formalize this argument in the proof of Lemma 2.2.

Once we have (CF), we obviously have that the reader returns a value in the domain. The reading function now is defined as follows: Suppose that the track-number that the high-level reading r uses is obtained from the subregister A by the subreading r_0. Let π_A be the reading function of A. We define $\pi(r)$ to be the high-level writing of which the subaction $\pi_A(r_0)$ is a part. The existence of such a high-level writing follows from Lemma 2.1.

It is now trivial to check that condition (F) of the atomicity criterion is satisfied.

Lemma 2.3. Condition (P) of the atomicity criterion is satisfied.

Proof. First observe that the furthest away a layer is, the more recent the value it carries. Condition (P) now follows immediately from the fact that if the reader, during a high-level reading r, obtains its track-number from the subregister A_l, then (regardlesss of whether a backup occurred or not) before the beginning of r, A_{l+1} was empty. In other words, there was no completed writing that used a layer further than l. The claim that A_{l+1} was empty is obvious in the case of a backup from $l+1$ to l. In the case of no backup, in other words if the reader has not yet advanced to $l+1$, it follows from the fact that before the beginning of r, RQ_l had no P (therefore the writer cannot advance to A_{l+1}).□

Lemma 2.4. Condition (I) of the atomicity criterion is satisfied.

Proof. Let $r_1 \rightarrow r_2$ be two high-level readings. Obviously, r_2 gets its track-number either from the same layer as r_1 or from one further ahead. In the second case, condition (I) follows from the fact that the furthest away a layer is, the more recent the value it has. In the first case, condition (I) follows from the atomicity of the subregisters A_l.□

The proof above is the second and last point we used the atomicity assumption. Figures 3 and 4 below depict all possible scenaria for the reader and the writer at the switch.

Proof of Lemma 2.2. We first prove two claims:

Claim I. The reader, during an execution of step (3) of its protocol, regardless of whether it backs up or not, starts reading a track-number from a subregister A_l *after* the writing of P has been completed on RQ_l.

Proof of Claim I. The claim easily follows, by the 'else' clause of step (3), when there is no backup. Also, when there is a backup, the reader backs up to a layer where a P has already been written. This follows from the remarks preceding the statement of Lemma 2.1. That proves Claim I.

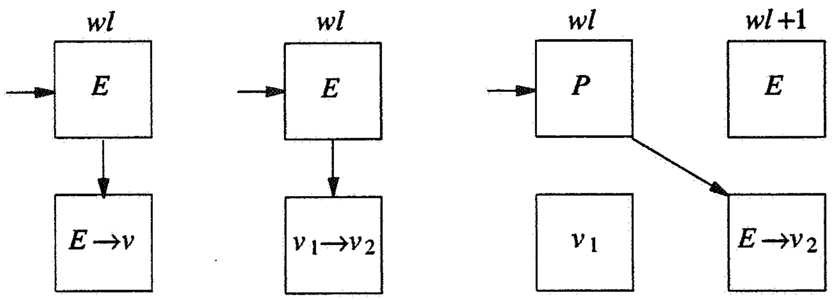

Figure 3: The three possible actions of the writer.

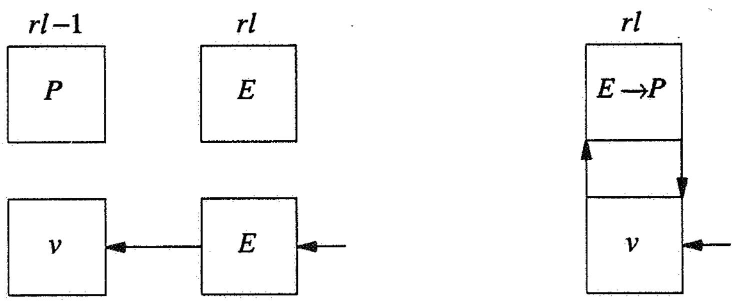

Figure 4: The two possible actions of the reader.

We now prove that:

Claim II. Let w be a high-level writing that during the execution of step (1) of the writer's protocol, reads from a subregister RQ_l the value P. The reader reads from A_l (during an execution of a step (3) of reader's protocol) a track-number written by either w^- or w^{--}.

Proof of Claim II. Let w_P be the subaction of w that reads P from RQ_l. Let r_P be the subaction by the reader that writes this (unique) P. Also, let r_t be a subaction by the reader when the track-number is obtained from A_l, during an execution of step (3) of the reader's protocol. We assume that w^{--} writes on A_l, because otherwise, the claim is trivial. By the previous claim, $r_P \rightarrow r_t$. We claim now that w^{--} must come to an end before the end of r_P. Indeed, otherwise, $r_P \rightarrow w^- \rightarrow w$, and therefore, w^- would read P from RQ_l and advance to the layer l+1. But then w would not read from RQ_l. So, indeed, w^{--} ends before r_P does. Because now $r_P \rightarrow r_t$, we have that $w^{--} \rightarrow r_t$. Therefore, by the regularity of A_l, r_t cannot read a value by a write preceding w^{--}. That completes the proof Claim II.

From the above we have that if at an instant t, the writer (during a high-level writing w) completes the reading of P from RQ_l, then at t, the reader can only be at one of the tracks the writer was at its last two steps. Indeed, at t, the printing of P on RQ_l must have started, so the reader has already advanced to layer l. Also, at t, further layers are still empty, so the reader can only be at a track whose number is obtained from A_l and therefore, by the previous claim, we obtain the required. Moreover, observe that, by the writer's protocol, immediately after the advance $wl := wl$+1 is executed, the two track-numbers where the reader may be are put into the forbidden set vb.

Similarly, if at an instant t, the writer reads E from RQ_l, the reader can only be at instant t

at a track whose number was obtained from A_{l-1}. But these are the track-numbers that were put into the set vb immediately after the advancing to l was executed.

So, the writer, according to the way the set vb is updated, at each high-level writing, chooses a track different from its last one (which is the one forwarded to the reader) and different from the tracks where the reader can be. This shows that the reader and the writer can never simultaneously be at the same track. That completes the proof of the Lemma 2.2. and the proof of correctness.□

2.3. Finitely Many Layers with Safe Subregisters.

In this section we suitably modify the protocol so that the subregisters A_l need only be regular and finitely many layers suffice.

First observe that in the proof of the previous section we did not assume that the subregisters RQ_l are atomic. Intuitively, the reason we did not use such an assumption is that once the reader reads the value P from a subregister RQ_l, it never visits this subregister again. Since RQ_l can only have two values, there is no way to have a new-old inversion. But we did use the atomicity assumption for the subregisters A_l. Informally, this assumption guarantees that in case of a backup to a subregister A_l, the reader does not read a value older than the value it read at its previous visit to A_l. Notice though that by Claim II of Lemma 2.2, the reader reads from a subregister, at any visit, one among two consecutive values. Suppose now that all values have an additional field (the tag) that ranges in $\{0,1,2\}$ and is incremented at each high-level writing by one modulo three. Using this tag the reader can, without the atomicity assumption, determine which among two consecutive values is the more recent one and thus avert new-old inversions. Below, we formalize in detail the above intuitive argument and in the sequel we attack the problem of finitely many layers.

In the previous section, the atomicity assumption of the subregisters A_l (apart from their regularity) was only used in order to have the following two properties.

A1. The reader, if during an execution of step (3) of its protocol is not instructed to back up, then, at this step, stores a value $\neq E$ into rt.

A2. Let v_1 and v_2, respectively, be the two values stored in rt at two executions of step (3) of the reader's protocol during both of which the reader reads from the same subregister. Then there is no new-old inversion between the v_1 and v_2.

In (A2) above, a new-old inversion between two values is defined in terms of the subwritings that write these values and the order that they are stored into rt (which determines the order of the corresponding visits to the tracks). The subwritings that write these values are determined from the subreadings that first read these values and the reading function of the corresponding subregister. Notice that, without the atomicity assumption, an inversion can only take place for executions of step (3) that read from the same layer. This is so, because the furthest away a layer is, the more recent its value.

Our objective now is to suitably modify the protocol so that on the one hand conditions (A1) and (A2) are satisfied without assuming atomicity and on the other hand, a run according to the modified protocol is a legitimate run according to the old protocol as well. Obviously then, all the switch-subregisters need only be regular. Indeed, our correctness proof for the previous protocol would also apply to the modified one.

To guarantee that condition (A1) is satisfied is fairly easy. We require that whenever the reader reads a value $\neq E$ at step (2) of its protocol (therefore, no backup will take place at the subsequent step (3)), it stores it at a local variable s. Now, if at the subsequent step (3) it reads the value E, then we require that it stores into rt the value of s rather than E. Formally, we introduce a local variable s, which initially is assumed to have no value, and the reader's protocol is modified by replacing step (3) of the reader's protocol by the following:

```
3a.  **if** $A_{rl} = E$ **then do** $rl := rl - 1$; read $A_{rl}$ and store its value into $rt$ **od**
3b.  **else do** /*$A_{rl} \neq E$*/
          store the value of $A_{rl}$ into $s$;
          write $P$ on $RQ_{rl}$; read $A_{rl}$;
          **if** $A_{rl} \neq E$ **then** store the value of $A_{rl}$ into $rt$ **else** $rt := s$ **od**;
```

Since, the first high-level reading is by assumption preceded by a high-level writing, s gets a nonempty value at the first high-level reading, and obviously it remains nonempty from then on. So condition (A1) is satisfied for the modified protocol. It is easy to see that a run according to this modified protocol is legitimate according to the old protocol as well. Indeed, the only possibility to have a value $\neq E$ at step (2) and then an empty value at the subsequent step (3), is that the writing of the $\neq E$ value is concurrent with the subreading of step (3). But then it is legitimate, according to the old protocol as well, to store into rt this nonempty value. That takes care of (A1).

We come now to (A2). For notational convenience, denote by $Pr1$ the protocol of the previous section and by $Pr2$ the protocol we get after the above modifications. We first give a lemma that is true for a run according to $Pr1$ (without the atomicity assumption) and therefore for a run according to $Pr2$ as well.

Lemma 3.1. Let v_1 and v_2, respectively, be the two values stored in rt at two executions of step (3) of the reader's protocol during both of which the reader reads from the same subregister (say, A_l). Then, v_1 and v_2 are consecutive or identical. In other words, there are two low-level writings w and w^+ such that the second directly precedes the first and such that both v_1 and v_2 are written by (one or two of) these subwritings.

Proof. Notice that the lemma does not claim that there is no new-old inversion between v_1 and v_2. It only claims that these values are consecutive or identical. We omit some of the details of the proof, since it is almost the the same as the proof of Claim II of Lemma 2.2. If the two values v_1 and v_2 are more than one subwriting apart, then the subwriting that wrote the more recent one (say, the v_2) belongs to a high-level writing that necessarily read P from RQ_l. This is so, because this high-level writing must have started after the end of the subreading that obtained v_1 (this can be easily seen by taking into account the interval geometry of the related durations and the regularity assumption). Therefore, v_2 will be necessarily written at a layer $\geq l+1$, a contradiction.□

Notice that because we do not assume atomicity, one or even both of the v_1 and v_2 above may be equal to E. The subwritings in this case refer to the initializing subwriting, which writes E. This is the reason that we avoided to consider the writings w and w^+ as high-level.

The protocol $Pr2$, although it satisfies (A1), may give rise to new-old inversions between values stored into rt. These can only be due to executions of reader's step (3) during which the subreadings are done on the same subregister. Observe that if we have a succession of executions of step (3) that read from the same subregister, then the first such execution involves no backup (i.e., clause (3b) is executed) while all other executions involve a back up (i.e., clause (3a) is executed). If $Pr2$ gives rise to new-old inversions when storing values into rt, then we can undo such inversions by instructing the reader to compare the value it gets each time from A_l (at an execution of step (3a)) with the value it had previously stored into rt. The sorting of the values to new and old can be effectively carried out by letting the reader compare an extra field (the tag) that all values have. As we have previously stated, the tag takes the values 0, 1 and 2. During step (5) of the writer's protocol, it is written, together with wt, on the subregister A_{wl}. At each high-level writing, it is incremented by one modulo three. Thus, the range of the subregisters A_l is $\{E\} \cup (\{0,1,2\} \times \{1,2,3,4\})$, i.e., its cardinality is thirteen.

Moreover, we assume that each processor, in order to handle the tag, has a new local variable. It is denoted by wtg for the writer and rtg for the reader. Both range in the set $\{0,1,2\}$ and are initialized by 0. The writer's protocol is modified by substituting the following step for the original step (5).

```
5.   $wtg := wtg+1 \pmod 3$; write $(wtg, wt)$ on $A_{wl}$.
```

The reader's protocol is modified by substituting the following for step (3) of the protocol $Pr2$.

```
3a.  **if** $A_{rl} = E$ **then do** $rl := rl-1$; read $A_{rl}$;
          **if** ($A_{rl} \neq E$ and tag of $A_{rl} = rtg+1 \pmod 3$) **then** store value of $A_{rl}$ into $(rtg, rt)$ **od**
3b.  **else do**
          store the value of $A_{rl}$ into $s$;
          write $P$ on $RQ_{rl}$; read $A_{rl}$;
          **if** $A_{rl} \neq E$ **then** store the value of $A_{rl}$ into $(rtg, rt)$ **else** $(rtg, rt) := s$ **od**;
```

Let us call $Pr3$ the protocol obtained by the above modifications. Notice that apart from the commands handling the tags, the two protocols differ only in reader's step (3a). It is also obvious that during a run of $Pr3$ we never store the value E into rt (the initializing values and the assumption of the existence of a high-level writing preceding all other actions guarantee this for the first high-level reading). We claim now that:

Lemma 3.2. A run by $Pr3$ satisfies (A2). Moreover, if we ignore the tags, such a run is a legitimate run according to $Pr2$ as well (and therefore it satisfies (A1)).

Proof. We prove the claim inductively on the number of reader's executions of step (3) that read from the same subregister, say, A_l. The only difference between $Pr2$ and $Pr3$, apart from the tags, is that in (3a) of $Pr3$, we store a value only if it is $\neq E$ and its tag $= rtg+1$, while in $Pr2$ we always store. Suppose therefore inductively that up to the k-th execution of (3) that reads from A_l, there are no new-old inversions. Suppose also that the values obtained up to this point, ignoring the tags, are legitimate according to $Pr2$ as well. Let v_2 be the value that the command 'read A_l' of clause (3a) returns during the k+1-th execution of (3). The fact that the k+1-th execution of step (3) involves a backup (i.e., clause (3a) is executed) follows from the assumption that the preceding k-th execution of step (3) reads from A_l as well. By the inductive hypothesis, the values that rtg and rt have at the beginning of the k+1-th execution of step (3) are legitimate according to $Pr2$ as well. Therefore, the value v_2 can be stored into rt according to $Pr2$ as well (but v_2 may precede (rtg, rt)). Since now both the k-th and the k+1-th execution read from A_l, Lemma 3.1 applies for v_2 and (rtg, rt). So, v_2 is more recent than (rtg, rt) if and only if $v_2 \neq E$ and the first coordinate of $v_2 = rtg+1$. If, on the other hand, $v_2 = E$ or if the first coordinate of $v_2 \leq rtg$, then v_2 precedes or is identical with (rtg, rt). Therefore, in this last case, if the two values are different, the duration of 'read A_l' of the k+1-th execution of (3) is concurrent with the subwriting that wrote (rtg, rt) (because a run by $Pr2$ is regular). So, in this case, it is legitimate according to $Pr2$ *not* to update (rtg, rt), in other words, not to store into it the value v_2. But that is what $Pr3$ does. Moreover, thus, a new-old inversion is averted.□

```
The writer writes a word $v$:
1.   read $RQ_{wl}$;
2.   if $RQ_{wl} = P$ then do $wl := wl+1$; $vb := \{wt, wt^-\}$ od;
3.   $wt^- := wt$; $wt := \min(\{1,2,3,4\}-(\{wt^-\} \cup vb))$;
4.   write the bits of $v$ onto the track with number $wt$;
5.   $wtg := wtg+1$; write $E$ on $A_{wl+1}$; /*the writer clears one layer ahead before writing*/
     write $(wtg, wt)$ on $A_{wl}$.

The reader returns a word:
1.   $rl := rl+1$;
2.   read $A_{rl}$;
3a.  if $A_{rl} = E$ then do $rl := rl-1$; read $A_{rl}$;
         if ($A_{rl} \neq E$ and tag of $A_{rl} = rtg+1$ then store value of $A_{rl}$ into $(rtg, rt)$ od
3b.  else do
         store the value of $A_{rl}$ into $s$;
         write $E$ on $RQ_{rl+1}$; /*the reader clears ahead one layer before printing a $P$*/
         write $P$ on $RQ_{rl}$; read $A_{rl}$;
         if $A_{rl} \neq E$ then store the value of $A_{rl}$ into $(rtg, rt)$ else $(rtg, rt) := s$ od;
4.   read the bits of track with number $rt$ and return the word thus obtained.
```

Figure 5: The final protocol (arithmetical operations are modulo 3).

The only thing left to be taken care of is the assumption of the unboundedly many layers. Informally, it can be checked from the protocol that at any instant during a subreading of the reader from a switch-subregister at layer l, the writer must have already started writing on the layer $l-1$ (otherwise, the reader would not have advanced to layer l). Moreover, the writer, cannot have advanced to a layer $>l+1$, because the reader has not yet written a P on RQ_{l+1}. That means that the reader and the writer stay at adjacent or concurrent layers at the subregisters A_l. Analogous things are true for the subregisters RQ_l. Therefore, to use only finitely many layers we employ the following trick. Assume that only three layers are available. Assume also that the arithmetical operations on rl and wl (subtraction or addition of 1) are done modulo three. But then we can run into a situation where a subreading advances one layer ahead of a subwriting and thus gets a wrong value, which exists at this layer because of a previous passing of writings over the layers. To avoid this situation, we further assume that the protocol before any subwriting at a layer l (either by the writer on A_l or the reader on RQ_l) instructs the processor to print the value E onto the corresponding subregister of the layer $l+1$. The assumption that we have three layers and the fact that the processors move side by side guarantees that this printing of E does not interfere with a noncompleted subreading at a layer behind.

In Figure 5, we give the final protocol that not only incorporates the modifications necessary for the assumption of regularity, but also formally describes the last modifications that make three layers suffice. We do not formally prove in this section that this recycling of layers is correct, since we do that in great detail in the algorithm of the last section, where actually the counting argument for the layers is more complicated.

Lamport (1986) gives an implementation of a 1W1R regular register that can hold m values from $m-1$ 1W1R safe, boolean (one-bit) registers. Using that result we have:

Theorem. An atomic, one-writer, one-reader, b-bit register can be implemented from $4b+39$ safe, 1-writer, 1-reader, boolean registers via the 4-tracks protocol.

3. Implementing a 1-writer, n-reader, b-bit Register.

By the previous construction (or by Lamport's (1986) construction to this effect), in order to implement an atomic 1WnRbB register from safe 1W1R1B subregisters, it suffices to give an implementation of the atomic 1WnRbB register by atomic 1W1R$f(n,b)$B subregisters, where f is an integer valued function. In the construction given below, $f(n,b)=O(n+b)$. Obviously, in any such implementation, in order for the writer to write a word to the compound register under construction, it must send a message to each reader by writing onto separate subregisters earmarked for each one of them. But then, the problem of new-old inversion may arise. Namely, if the writing of an old value v_{old} has been completed and the writing of a new v_{new} is only partially carried out (i.e., the writer has written the message corresponding to v_{new} only to some of the subregisters earmarked for the readers) then it is possible that a reading r_1, that finishes before another reading r_2 starts, returns v_{new}, while r_2 returns v_{old}. Essentially, this is the only real problem that must be faced in such an implementation. We solve this problem by a) having a reader write the word it returns into separate subregisters earmarked for each one of the readers, and b) having each reader read, apart from the subregister it shares with the writer, also all the subregisters it shares with the other readers. The crux of the argument is that, in this way, r_2 will see the new value returned by r_1, since r_1 writes this value onto the joint-communication register.

3.1. Architecture.

The architecture we use to accomplish this kind of one-to-one communication between all processors involved is a matrix similar to the one used in Vitányi and Awerbuch (1986). Each processor is assigned a column of subregisters that it may use to receive messages from the other processors. It is also assigned a row of subregisters that it may use to communicate messages to the other processors. If the writer is considered to be the processor numbered 0 and the readers the processors numbered 1 through n, then the matrix has dimensions $(n+1)\times(n+1)$ (with rows and columns numbered from 0 to n) and the subregister at row i and column j can be written by the processor i and can be read by the processor j. Observe that the diagonal elements of such a matrix are useless (because there is no need for a processor to communicate with itself).

Nevertheless, for notational simplicity, all of them except the one on the left top corner are included. The problem now faced is how the reader can sort the most recent message from the ones it receives when reading the items of a column. We solve this problem by a) having the processors write a suitable tag, and b) having not only one matrix but many of them lying in different layers. Roughly, the idea behind the layers is the following: Once a reader starts reading a value, instruct the writer to execute all subsequent writings onto a further layer. That way, the values read by the reader belong to a set of C consecutive ones, where C is a small constant. The tag is used in order to handle the layers and to find the most recent one among the values considered. As is easy to see, this is the same idea as the one employed in the previous construction, where the layers corresponded to the array of pairs of subregisters (A_l, RQ_l). Actually, we need layers only for the columns numbered 1 through n. Specifically, the architecture of our implementation comprises the following (Figure 6):

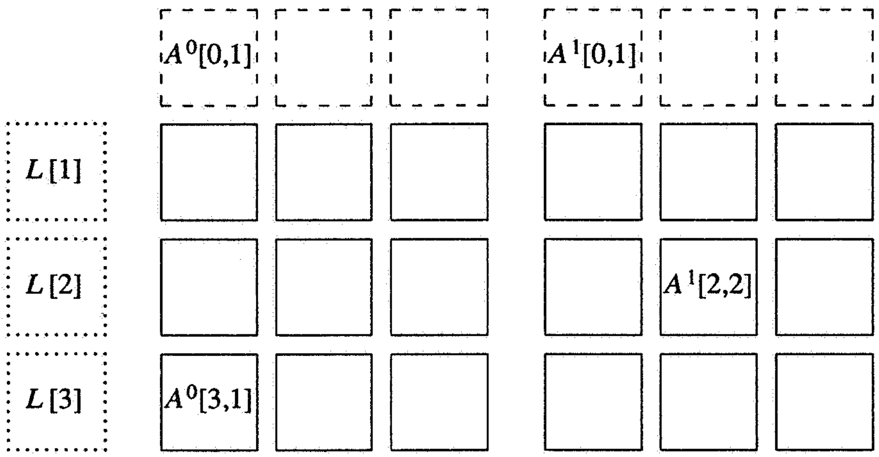

Figure 6: The L-column and two layers of the 1-writer, 3-reader, atomic register.

a) A column $L = (L[1],..., L[n])$ of n subregisters (dotted boxes forming the leftmost column in Figure 6). This is the column which, in the discussion above, was referred to as the column numbered 0 (apart from its diagonal element). So, each subregister $L[i]$ can be written by the reader i and read by the writer.

b) A sequence $(A^l)_l, l = 0,..., c$ of $(n+1)\times n$ matrices each consisting of $n(n+1)$ subregisters. These comprise the layers of the columns 1 through n that were mentioned above. We refer to the subregisters in the matrix A^l as the subregisters at layer l. Two such layers are depicted in Figure 6. For any layer l, the subregisters in the row $(A^l[0,1], \ldots, A^l[0,n])$ (dashed boxes) are written by the writer, while the ones in the row $(A^l[i,1], \ldots, A^l[i,n])$ (solid boxes) are written by reader i $(i = 1, \ldots, n)$. In addition, the j-th reader can read the entries of the j-th column $(A^l[0,j], \ldots, A^l[n,j])$. For our implementation, seven such layers suffice (i.e., $l = 0,..., 6$). So, altogether, we need $7n(n+1)+n$ atomic 1W1R$O(n+b)$B registers. However, to make the presentation clearer, we first suppose that an unbounded number of layers is available. We give the protocol of the implementation under this assumption and then, observing that both the reader and writer move within a bounded 'window' of layers, we show that only seven layers are enough. Intuitively, this is so because a layer that has come out of the scope of the window can be recycled. According to the protocol, numbers referring to the layers will be written to the subregisters. Therefore, the assumption that the number of layers is unbounded forces us to assume that the subregisters can hold a value from an infinite domain. This has no real consequences though, since, finally, only seven layers will be utilized.

3.2. The Algorithm for Unboundedly Many Layers.

All the subregisters involved in the construction are assumed to be atomic. Therefore, the low-level actions on them can be considered instantaneous. In the code below (Figures 7 and 8), we assume that there is an unbounded number of layers numbered 0, 1,....

The writer has an array of local variables $(l_1, \ldots, l_n)$ that are used to decide which layers to write to. The variables l_i are initialized by the value 0. Also, it has a local variable s that takes values 0, 1,..., 4 and is incremented each time by 1 (mod 5). It is initialized by the value 0. The reader has a local variable l that is used to decide the layer where it will read. It is initialized by the value -1 (a reading starts by setting $l := l+1$). Moreover, the processors have other local variables, described in the code, and arbitrarily initialized. They are used to store values to be used later.

We now come to the subregisters (shared variables). The $L[i]$'s contain a layer-number. They are initialized by the value 0. All other subregisters $A^l[i,j]$ carry either a value denoted by E and indicating that no writing action has yet taken place on them or a value consisting of the following two fields: a) A tag that consists of a tuple of the form $(s, (l_1, \ldots, l_n))$, where the s and the l_i's are the values of the corresponding local variables of the writer. We find it better not to unnecessarily load the notation by distinguishing the symbols for the local variables and for the components of the tags. b) An element from the domain of the compound register i.e., a word. All the subregisters $A^l[i,j]$ are initialized by the value E. Subregisters with the value E are called empty. We assume that the initial value of any subregister used in a run is written by an initializing subaction that precedes all other subactions on this subregister (thus, the reading functions on the atomic subregisters will always be defined). The initializing subactions are not considered parts of high-level actions. We also assume that there is a high-level writing of a word $\neq E$ which precedes all other high-level actions (this is not considered an initializing action).

For notational convenience, let $\bar{l} = (l_1, \ldots, l_n)$ and $\overline{l+1} = (l_1+1, \ldots, l_n+1)$. Also, let $A^{\bar{l}}[.,i]$ be $(A^{l_1}[1,i], \ldots, A^{l_n}[n,i])$, i.e., the i-th column apart from the element at the top and with the layers chosen according to $\bar{l}$. Finally, let $A^{\bar{l}}[0,.]$ be $(A^{l_1}[0,1],\ldots,A^{l_n}[0,n])$, i.e., the top row.

Writer writes v to the compound register:

```
1.  for $i := 1$ to $n$ do $l_i := L[i]$ od;
2.  $s := s+1 \pmod 5$;
3.  for $i := 1$ to $n$ do write the tag $(s, \bar{l})$, as well as the value $v$ in $A^{l_i}[0,i]$ od.
```

Figure 7: The writer's protocol.

Reader i returns a word from the the compound register:

```
1.  $l := l+1$;
2.  read $A^l[0,i]$; if $A^l[0,i] = E$ then $l := l-1$;
3.  $L[i] := l+1$;
4.  read $A^l[0,i]$, call $(s_c, \bar{l^c})$ the tag and $v_c$ the word obtained and store them into local vari-
    ables;
5.  read $A^{\bar{l^c}}[.,i]$ and $A^{\overline{l^c+1}}[.,i]$ and store the values of their components into local variables;
6.  read $A^{l+1}[0,i]$ /*possibly for the second time*/;
7.  if $(A^{l+1}[0,i] = E)$ then invoke procedure select
    procedure select:
      (i)  determine whether among items stored in step (5) there is one with a value $\neq E$ and
           with the $s$ in its tag equal to either $s_c+1$ (mod 5) or to $s_c+2$ (mod 5);
      (ii) if the answer to (i) is 'yes' then return the word of any item having an $s$ as described
           else return $v_c$
    end select
    else do $l := l+1$; $L[i] := l+1$; read $A^l[0,i]$ and return its word od; /*There is a high-level
    writing that writes on $A^{l+1}[0,i]$ and overlaps the current high-level reading.*/
8.  for $k := 1$ to $n$ do write the word returned and its associated tag to the subregister $A^l[i,k]$
    od.
```

Figure 8: The reader's protocol.

We now informally outline what the protocol does. First, notice that any non-initial value appearing on any subregister has been originally written on a (possibly different) subregister by a high-level writing. Therefore, with every value that is returned by a subreading, we can associate a high-level *writing* that originally wrote this value. However, since different subwritings can write equal values, the high-level writing associated with a value is not uniquely defined. This indeterminacy is resolved by the definition, in the correctness proof below, of the reading function π. In the informal discussion below, we assume that with every value we can uniquely associate a high-level writing that originally wrote it. Also, we say that a value is one high-level writing behind a second value, if these values were originally written by two successive high-level writings, respectively.

The writer, in order to write v to the compound register, writes v together with a tag to the components of the upper row of the matrix. These components are the subregisters where the writer communicates with each reader. The layer-number for each is obtained by reading the shared registers $L[i]$. Notice that the communication between the writer and *each* reader is carried out at different layers. The objective of this is to have the writer and each reader always be at the same or adjacent layers of the subregisters they share. The writer, at each high-level writing, increments the value of its local variable s by one modulo a small constant, say C. The tag that it writes (which, like v, is the same for all the components it writes to) consists of the value of s and all the layer-numbers for the current writing. The s is written in order to enable a reader determine the most recent one among the values that it considers. These values, as we will see, belong to a set of C consecutive ones. The layer-numbers are written in order to inform each reader about the layer where the writer communicates with the other readers.

The reader i, during a high-level reading r, reads first the subregister where it communicates with the writer. It chooses the layer which is next to the one given by its local variable l (by setting $l := l+1$). If the reader, because of this original advance to the next layer, finds the subregister that it shares with the writer empty, it backs up to the previous layer (i.e., decreases the value of l). Otherwise, it does not change layer. Thus, it will always be at a layer with a nonempty value. Next, it prints the current layer-number (i.e., the current value of l) incremented by one at $L[i]$. It does the same after any other advance (i.e., incrementing of l) that it may perform during r. Thus, after an advance of the reader, later writings by the writer, will also advance a layer at the joint subregister. The mode of advancing of the reader and the writer guarantees that the writer and each reader are always at the same or adjacent layers of the subregisters they share. But the important consequence of the obligatory advancing of later writings to further layers is the following:

a. If at an instant during step (8) of the protocol the reader i writes a value on $A^l[i,k]$, then this value is at most one high-level writing behind the value on $A^l[0,i]$. Moreover, this remains true at any later instant.

To accomplish this, the reader must obtain a value from the writer *after* it has changed $L[i]$. Because, otherwise, a fast writer can, before $L[i]$ is changed, execute many writings. So, the reader reads (possibly for the second time) the subregister it shares with the writer. It keeps this value as a **candidate** value to return. From this moment on, it is only the high-level writing that directly follows the candidate that could write on $A^l[0,i]$. And that only in case this writing checked $L[i]$ before its updating by the corresponding reader.

From the tag of the candidate value, the reader gets the layer-numbers for the components of the i-th column, where it communicates with the other readers. It reads all the values there, as well as the ones at the next layers and stores them. Observe that if an earlier reading returns a value later than the candidate, this value will be the one immediately after the candidate. Therefore, it will be written by the earlier reading in one of the columns examined. This is so because the protocol guarantees that the following is true:

b. If r returns the value of a writing w, then at the last step of its protocol, r writes this value on the i-th row and chooses the layer that is given by the i-th coordinate of the layer vector of either w or w^-.

After this polling of the other readers, the reader reads again the register where it communicates with the writer, but at the next layer. This register, at the beginning of r, was empty. If it is empty again, up to this point, writings that are ahead the candidate at most two high-level writings may have started writing on the upper row. This is so, because of the obligatory advancing of the writer to a further layer. Also, on the upper row, at the layers examined, we cannot have values that are more than one high-level writing behind the candidate value. This is so, because the candidate writing has already started. Now, by (a), the messages the readers communicate are at most one high-level writing behind the corresponding values of the writer. Therefore, the messages received by the current reading from the other readers can at most be a small constant number of high-level writings away from the candidate value. So, the reader can sort these messages by using the value of s (which is incremented by one modulo a small constant at each high-level writing). Thus, it can avert new-old inversions.

If, on the other hand, the register at the next layer is now found nonempty, its value must have been written by a high-level writing that overlaps r. If a reading returns the value of an overlapping writing, then there can be no earlier reading that had returned a more recent value. Therefore, in this case, the reader may ignore the values received from the other readers. So, the reader advances one layer ahead, instructs subsequent writers to move onto the next layer (by incrementing $L[i]$), re-reads the now nonempty register and returns its word. The re-reading *after* the incrementing of $L[i]$ is done in order to make (a) true. In each case, at the end, the reader writes the value it returns to the row i at the layer l. It can be seen that thus, (b) remains true.

3.3. Correctness.

3.3.1. Preliminaries.

As we have already stressed, the subregisters are atomic. Therefore, the subactions on them can be assumed to be instantaneous. Thus, we suppose that with every action on a subregister we have associated a positive real number t, which is the instant that this subaction takes place. Also, subactions on local variables can be assumed to be instantaneous, since they are executed sequentially by the corresponding processor. We associate a time instant with them too. For notational simplicity, when no confusion may arise, we will denote a subaction and the instant it takes place by the same symbol. It is convenient to assume that no instant in time harbors two different subactions (on shared or local variables). We call the real numbers associated with subactions (on shared or local variables) **significant** instants. Since we have assumed that for any action a of a run, the set $\{b: \text{not}(a \rightarrow b)\}$ is finite, and since a high-level action contains only finitely many subactions, we have that the set of significant instants of a run is either finite or is unbounded and has the order type of the positive integers. Thus, induction on the instant that a subaction takes place can be carried out. Also, we have that for any significant instant t, there is an open interval I_t that contains t and such that no significant instant other than t lies in I_t. If t is any significant instant, the expression '**immediately after** t' means at any instant $t' > t$ such that $t' \in I_t$. Thus, between t and t', no subaction occurs. We interpret the expression '**just before** t' similarly.

We now come to the reading function π of the compound register. Since every subregister A is atomic, we have a total reading function π_A on each subregister. Given a high-level reading r, let r_0 be the low-level reading that reads the value returned by r. Suppose that r_0 takes place on A. Consider $\pi_A(r_0)$. If this is not an initializing subwriting, it belongs to a high-level action. Call this, if it exists, $\gamma(r)$. Now, $\gamma(r)$ can be a high-level reading (which writes the value it returns). In this case, we consider (if it exists) $\gamma(\gamma(r))$, and so on. This procedure cannot go forever, because then we would have an infinite descending chain of significant instants. Therefore (unless at some point we hit an initializing subaction), we will eventually obtain a high-level writing. This defines $\pi(r)$. In the proof below, we will show that our initialization conditions guarantee that π is always defined. Notice that by the same repetitive procedure, we can associate a *high*-level writing not only with each high-level reading but also with each *low*-level reading. We assume therefore that the function π applies both to high- and low-level readings. In the proof below we shall show that conditions (P) and (I) of the atomicity criterion are satisfied. Condition (F) needs no proof, since a high-level reading returns a value chosen among the values associated with its subreadings. Therefore, condition (F) for the compound register follows from the same condition on the subregisters. Notice though that condition (P) needs to be proved.

Indeed, *prima facie*, a reading may return a value from a past layer, while a completed, more recent high-level writing may have used a further layer.

3.3.2. Proof.

The proof will be given by a series of lemmata. We fix a reader i. Step numbers from now on refer to the steps of the reader's protocol. It is immediate to check that every time the local variable l of the reader changes, it is either increased or decreased by one. We refer to increasings of l as **advances** and to its decreasings as **backups**. Observe that there are no consecutive backups, because any backup that may take place at step (2) is preceded by step (1), where an advance takes place. As a consequence, the value of l at any instant, not only is $\geq$ its last value minus one, but also, is $\geq$ all its values preceding the last one. This can be formally seen by examining the partial sums of a sequence whose terms are either 1 or -1 and has no consecutive negative terms. Also, observe that once a subregister becomes nonempty, from then on it stays nonempty.

Lemma 3.1. The value of the shared variable $L[i]$ is non-decreasing in time. Actually, each time that it increases, it increases by one. Moreover, at any instant, $l \leq L[i] \leq l+1$. Actually, immediately after setting $l := l+1$, we have that $L[i] = l$ and, of course, immediately after setting $L[i] := l+1$, we have that $L[i] = l+1$.

Proof. If we had no backups, the claim would have been obvious. Indeed, assuming no backups, after each time l is incremented, $L[i]$ is set equal to $l+1$ and that takes place before the next change in l. Moreover, those are the only changes on $L[i]$. So, in the absence of backups and by the given initialization of the variables, l drags at most one behind $L[i]$. Nevertheless, even with backups, this is again the case. Indeed, consider the first ever backup by the reader i. It is preceded by an action $l := l+1$. However, between this incrementing of l (which makes l equal to $L[i]$, because there are no previous backups) and its decreasing at the backup, no change of $L[i]$ takes place. Therefore, as far as the value of $L[i]$ is concerned, we can ignore the backup, since it amounts to nothing more than one step forwards followed by one step backwards (these steps do not contradict the fact that l drags at most one behind $L[i]$). The claim now follows by induction on the backups. Notice that after a backup, step (3) does not change the value of $L[i]$.□

By the previous lemma, the writer either writes at the layer it wrote last or it advances one layer. We now prove a more interesting fact.

Lemma 3.2. Just before any instant that l is incremented by one, $A^l[0,i]$ is nonempty. Intuitively, that means that the reader never advances, unless the current layer is nonempty.

Proof. Consider an instant t_e, during a high-level reading, when $A^l[0,i]$ is empty, and suppose that at t_e, an advance takes place. Let l_e be the value of l at t_e. Obviously, $A^{l_e}[0,i] = E$ at all instants preceding t_e. First observe that $l_e \neq 0$, because the high-level writing that by assumption precedes all high-level actions writes on the upper row, at the 0-th layer. So, the reader advanced to layer l_e at a previous instant by incrementing the value that l had at that previous instant. Let t_a be the instant of the last advance to l_e that occurred before t_e. That advance cannot have taken place by an execution of the last part of step (7), because $A^{l_e}[0,i] = E$ at least until t_e. Therefore, it took place by an execution of step (1). But t_e is a significant instant which is not a backup. Also, $A^{l_e}[0,i]$ is empty up until t_e. Therefore, step (1) must be followed by an execution of step (2) that includes a backup. Since, at t_e there is no backup, we conclude that there is a backup that must be executed before t_e and after t_a. But then, another advance, later than the one at t_a, must take place in order to reach l_e, a contradiction. □

As a consequence, we have that immediately after a backup, $A^l[0,i] \neq E$. Indeed, in order to backup to a layer, we must have made an advance from it, so the required follows by the previous lemma. Therefore, at step (4), the reader always executes a subreading that returns a value $\neq E$. We call the high-level writing that writes this value the **candidate** writing, and we denote it by w_c (it varies with the current high-level reading, of course). As in the code, its value is denoted by v_c, and its corresponding tag by $(s_c, \overline{l^c})$.

By the previous remark and by examining the steps of the reader's protocol where a return takes place, we can see that with every high-level reading we can *always* associate a high-level action that writes the value that the reading returns. So, we have the following:

Corollary 3.1. The reading function π is always defined for high-level readings. Moreover, if a low-level reading is executed on a nonempty subregister, then the function π is defined for it (and returns a high-level writing). □

As a further consequence of the above lemmata we have that: a) Immediately after reader i increments l, for all $k \geq l+1$, $A^k[0,i] = E$. This is so because, at such an instant, by Lemma 3.1, $L[i] = l$. b) Immediately after reader i increments l, for all $k < l$, $A^k[0,i] \neq E$. This is a restating of the fact that no advance is possible from an empty layer (Lemma 3.2). c) There are no consecutive backups. These facts show that the reader i and the writer move on the registers $A^k[0,i]$, $k \geq 1$, within a window of two consecutive layers.

At the statements below, r denotes an arbitrary high-level reading and w_c the candidate writing associated with r.

Lemma 3.3. There is no w such that $w_c \rightarrow w \rightarrow r$.

Proof. Assume, towards a contradiction, that there is such a w. Let $l_{(4)}$ be the value of the local variable l at the instant that w_c is read, i.e., at the instant step (4) of r is executed (subscripts in parentheses refer to step numbers). By the previous lemmata, whether a backup occurred or not, just before the subreading of step (2) of r, $A^{l_{(4)}+1}[0,i] = E$. But that means that w, which precedes r, overwrites w_c on the subregister $A^{l_{(4)}}[0,i]$. Therefore, the subreading of r at step (4) would read w and not w_c, a contradiction.

An immediate consequence of the above is that once we prove that r always returns either w_c or a later write, we have that condition (P) is satisfied.

Lemma 3.4. Let $t_{(4)}$ be the instant that r, during step (4), reads $A^{l_{(4)}}[0,i]$ (and obtains v_c). There is at most one high-level writing that may write onto $A^{l_{(4)}}[0,i]$ after $t_{(4)}$.

Proof. By step (3) of the protocol, just before $t_{(4)}$, $L[i] = l_{(4)}+1$. Therefore, any high-level writing that starts after $t_{(4)}$ will choose, in order to communicate with reader i, a layer $\geq l_{(4)}+1$. Therefore, either $A^{l_{(4)}}[0,i]$ will never change value after $t_{(4)}$, or at most one high-level writing will write there after $t_{(4)}$. Namely, one that checks $L[i]$ before the first time it was set to $l_{(4)}+1$ and writes on $A^{l_{(4)}}[0,i]$ after $t_{(4)}$.□

Lemma 3.5. Suppose $j = 1, \ldots, n$. Let $t_{(8)}$ be an instant, when r writes a value on $A^{l_{(8)}}[i,j]$ ($l_{(8)}$ denotes the value of l at $t_{(8)}$). Then, at any instant $t' > t_{(8)}$ the value of $A^{l_{(8)}}[0,i]$ is at most one high-level writing behind the value written by r at $t_{(8)}$. Formally, that means that if at $t' > t_{(8)}$, a low-level reading r_0 reads from $A^{l_{(8)}}[0,i]$, then $\pi(r_0)$ either follows $\pi(r)$ or is identical to it or directly precedes it (notice that the value that r writes on $A^{l_{(8)}}[i,j]$ is the value that $\pi(r)$ writes on the upper row).

Proof. This lemma is equivalent to condition (a) mentioned in the informal outline of the protocol. The proof of the claim will be by induction on $t_{(8)}$. First notice that at t' the subregister $A^{l_{(8)}}[0,i]$ is nonempty. We will prove the claim only in the case that the returning of r is executed by step (ii) of the procedure 'select'. The other case, namely when the returning is executed by the last clause of step (7), is easier and needs nothing essentially different. Also, if the reading r, at the end of the procedure 'select', returns the candidate word, then the claim follows easily. This is so, because at any instant t' with $t' > t_{(8)} > t_{(4)}$, at the subregister $A^{l_{(4)}}[0,i]$, we will have either the candidate value or the one by w_c^+.

We suppose, therefore, that r returns a value selected at step (i) of the procedure 'select'. Let $t_{(6)}$ be the instant that step (6) of r is executed. Obviously, $t_{(4)} < t_{(6)} < t_{(8)}$. During 'select', the values of of $A^{\overline{l^c}}[.,i]$ and $A^{\overline{l^c+1}}[.,i]$ that are not E are examined. These have been obtained at step (5), i.e., during the interval $[t_{(4)}, t_{(6)}]$. Therefore, they were written before $t_{(6)} < t_{(8)}$. So, the induction hypothesis applies. Therefore, at the instants these values were obtained, the corresponding values on $A^{\overline{l^c}}[0,.]$ and $A^{\overline{l^c+1}}[0,.]$, can be at most one high-level writing ahead. Let us now see what values may we have on the rows $A^{\overline{l^c}}[0,.]$ and $A^{\overline{l^c+1}}[0,.]$, during the interval $[t_{(4)}, t_{(6)}]$. Call these rows, temporarily, first and second row, respectively (this numbering refers to the layers of the rows and not their position in the matrix). At $t_{(4)}$, we read w_c from $A^{l_{(4)}}[0,i]$ (which, because $l_{(4)} = l_f^c$, lies at the first row). At $t_{(6)}$, we read E from $A^{l_{(4)}+1}[0,i]$ (which lies at the second row).

We will now show that as a consequence, the values that may appear during $[t_{(4)}, t_{(6)}]$, on either of the two rows, are among the values written by w_c^-, w_c, w_c^+, w_c^{++}. Indeed, w_c executes all its subwritings on the first row. Therefore, any subwriting by a writing preceding w_c^- that is executed on the first row will be overwritten by w_c^- before $t_{(4)}$ (because the latter is completed at $t_{(4)}$). So, during $[t_{(4)}, t_{(6)}]$, we can have no value by a writing preceding w_c^-. On the other hand, by Lemma 3.4, on $A^{l_{(4)}}[0,i]$, after $t_{(4)}$, at most one writing following w_c can write. The w_c^{++} will have to move onto $A^{l_{(4)}+1}[0,i]$ or further. But $A^{l_{(4)}+1}[0,i]$ is empty at $t_{(6)}$. Therefore, by $t_{(6)}$, w_c^{++} is, at best, half-complete. Therefore, no values by a writing following w_c^{++} can appear on either row during $[t_{(4)}, t_{(6)}]$. That shows that during this interval, on either row, we can have, at most, values written by the four consecutive high-level writings w_c^-, w_c, w_c^+, w_c^{++}.

Therefore, by the inductive hypothesis, the values obtained from the columns $A^{\overline{l^c}}[.,i]$ and $A^{\overline{l^c+1}}[.,i]$ at step (5), are among the values written by the five consecutive high-level writings w_c^{--}, w_c^-, w_c, w_c^+, w_c^{++}. Since now s_c comes from w_c, s_c+1 (mod 5) and s_c+2 (mod 5) can only be written by w_c^+ and w_c^{++}, respectively. Therefore, we proved that if the writing that is returned by r is selected by an execution of step (i) of 'select', then it is a writing that follows w_c. Since the procedure 'select' is executed at step (7), we have that $l_{(8)} = l_{(4)}$. But, obviously, $l_{(4)} = l_f^c$ Therefore, since the reader writes on a row at the layer $l_{(8)}$, the claim is proved. □

We have actually proved a lot more. It can be easily seen now that in any case (whether we use 'select' or not) the writing that is returned, if it is not w_c, it follows it. So, by Lemma 3.3, we have the following:

Corollary 3.2. Condition (P) of the atomicity criterion is satisfied.□

By the proof of Lemma 3.5, we have also shown that if at step (5), a value by w_c^+ or w_c^{++} is obtained, and if the procedure 'select' is invoked, then one of the w_c^+, w_c^{++} is returned.

Lemma 3.6. Suppose that r returns w and that $\overline{l}$ and $\overline{l'}$ are the corresponding values from the tags of w and w^-, respectively. Then r writes (at step (8) of its protocol) the value that it returns onto either $A^{l_i}[i,k]$ or $A^{l_i'}[i,k]$, $k = 1,\dots, n$.

Proof. This lemma is equivalent to condition (b) mentioned in the informal outline of the protocol. We prove it inductively. The only point that needs some clarification is when r returns w_c^{++} at step (i) of 'select'. But then it is easy to check that the component l_i of the $\bar{l}$ of w_c^+ is $l_{(4)}$. This is the layer of the row where the reader writes. So, this claim is also proved.□

Lemma 3.7. Condition (I) of the atomicity criterion is satisfied.

Proof. Suppose that $r_{\text{old}} \rightarrow r$ and suppose that r_{old} is executed by processor j. If r returns its value by an execution of the 'else' clause of step (7), then the writing that it returns overlaps r. So we cannot have any inversion by r_{old} and r. Suppose, therefore, that 'select' is invoked. Let $w_{\text{new}} = \pi(r_{\text{old}})$ and suppose that $\pi(r) \rightarrow w_{\text{new}}$. Then it follows by (P) and (F) that w_{new} is concurrent with both r and r_{old}. By the remark preceding Corollary 3.2, we have that $w_c \rightarrow w_{\text{new}}$. If also $w_c^+ \rightarrow w_{\text{new}}$, then since the latter is concurrent with both r and r_{old}, it follows by the interval geometry of the durations of the actions involved that $w_c^+ \rightarrow r$. But this contradicts Lemma 3.3. So, w_{new} is w_c^+. But then, by Lemma 3.6, the reading r_{old}, at its last step (i.e., before r starts), will write the value of w_c^+ onto either $A^{l_f}[j,i]$ or $A^{l_f+1}[j,i]$. Therefore, the procedure select will certainly return a writing that is either w_c^+ or w_c^{++}. So, again no inversion has taken place.□

By the above lemma, the correctness proof is completed.

3.4. Finitely Many Layers.

In this section we prove that finitely many layers suffice for the previous construction. Once we assume that we only have a constant number of layers, then the incrementing of the readers' local variables l is done modulo this small constant. But then the protocol may run into one of the following problems:

a. In the case of infinitely many layers, processor i may read, at an instant t, from a subregister $A^k[j,i]$, a value $\neq E$. Moreover, *before* t, processor j may have executed a writing at a subregister $A^{k'}[j,i]$, with $k' > k$. But in the case of finitely many layers, the advancing of the processors is done modulo the number of layers. So k' and k may turn to be equal modulo this number and therefore, the writing on $A^{k'}[j,i]$ may interfere with the reading on $A^k[j,i]$, while it should not.

b. In the case of infinitely many layers again, processor i may read from $A^k[j,i]$ the value E. But in the case of finitely many layers, $A^k[j,i]$ may have a value which is $\neq E$, because of a previous passing of the processor j over the finitely many layers.

First notice that a reading may execute some subreadings the values of which will never be used. Those are the subreadings that are executed at step (5) of a reading r during which procedure 'select' is not invoked (i.e., the 'else' part of step (7) is executed). All subreadings not belonging to this category are called **useful**.

Fortunately, as we will prove, in the infinitely many layers case, at any instant during a high-level reading r by reader i and for any $j \geq 0$, the last subreading of r and the last subwriting by processor j that have been performed on $A[j,i]$ cannot be but a small constant number of layers apart. This is true under the assumption that this last subreading is a useful one. If this constant is C, then to avert problem (a) above, we have to have $\geq C+1$ layers. In order to avert problem (b), we require that immediately before the execution by processor j of a subwriting on a subregister $A^k[j,i]$ ($j \geq 0, i \geq 1$), the processor j prints E on all $A^{k+q}[j,i]$ with $q = 1,\dots, C$. This is called **clearing ahead**. But then we must take care that this clearing ahead does not interfere with possible future subreadings at layers behind (this interference with layers behind is possible in the finite case, because of the 'wrapping' of the layers onto themselves). But such a future subreading is again at most C layers behind $A^k[j,i]$. Therefore, having $\geq 2C+1$ layers suffices.

Actually, as we will see by a fine counting argument, seven layers and a clearing ahead of four layers is sufficient. Formally, the protocols for the finite case are obtained from the protocols of the infinite case by only the following three changes:

i. Interpret all arithmetical operations, except the ones on the variable s, as operations modulo seven.

ii. Replace step (3) of the writer's protocol by the following step:

```
3.  for $i$ := 1 to $n$ do for $q$ := 1 to 4 do write $E$ on $A^{l_i+q}[0,i]$ od; write the tag $(s,\bar{l})$, as
    well as the value $v$ in $A^{l_i}[0,i]$ od.
```

iii. Replace step (8) of the reader's protocol by the following step:

```
8.  for $k$ := 1 to $n$ do for $q$ := 1 to 4 do write $E$ on $A^{l+q}[i,k]$ od; write the word returned
    and its associated tag to the subregister $A^l[i,k]$ od.
```

Observe that apart from the different interpretation of arithmetic operations, the only difference from the infinite case is that all subwritings executed at subregisters associated with layers are preceded by a clearing ahead. These subwritings are executed only at the very last steps of the respective protocols.

To formally prove now the correctness of the above protocol, we first prove some lemmata that exactly tell us how far apart the current subreading and the current subwriting can be. These lemmata refer to a run according to the protocol for the infinite case.

Lemma 4.1. Let r be a high level reading by the reader i such that the procedure 'select' is invoked during r. Let, as usual, $(s_c, \bar{l^c})$ and v_c be the tag and the word, respectively, of the candidate writing w_c associated with r. Suppose that r, at step (5) of its protocol, reads, at an instant $t_{(5)}$, the subregister $A^k[j,i]$, for some $j \geq 1$ ($k = l_j^c$ or $k = l_j^c+1$). Consider a subwriting $W(r_j)$ of a high-level reading r_j by the reader j which has the property that all subactions of r_j that precede $W(r_j)$ are executed before $t_{(5)}$. If $W(r_j)$ is executed on $A^{k'}[j,i]$, then $k' \leq k+2$.

Proof. Intuitively, the lemma states that a subwriting by a reader cannot go more than two layers ahead the current subreading (under the assumption that this subreading is a useful one). Let $t_{(6)}$ be the instant that step (6) of r is executed. Obviously, $t_{(5)} < t_{(6)}$. Since after $t_{(6)}$, procedure 'select' is invoked, by the proof of Lemma 3.5, up until $t_{(6)}$, the latest high-level writing that may have executed a subwriting is w_c^{++}. The high-level reading r_j by j, according to the hypothesis, executed its subreadings before $t_{(5)}$. So, $\pi(r_j)$ (whose value is written by $W(r_j)$) can only be a high-level writing that executed at least one subwriting before $t_{(5)} < t_{(6)}$. Therefore, $W(r_j)$ can only write a value written by w_c^{++} or an earlier high-level writing. But then, the layer of $W(r_j)$ can be at most l_j^c+2. The lemma now follows from the fact that $k = l_j^c$ or $k = l_j^c+1$□

Lemma 4.2. Let r and i be as in the previous lemma (it is not necessary, in this case, to assume that 'select' is invoked during r). Suppose that r, at an instant t, reads from $A^k[0,i]$. Suppose, moreover, that the writer (processor numbered 0) executes a subwriting W on $A^{k'}[0,i]$. If all subactions of the writer that precede W have been executed before t, then $k' \leq k+1$.

The above lemma is the counterpart of Lemma 4.1 when the processor j is not a reader, but rather the writer. The bounds on the possible advance of the writer are stricter in this case. We do not formally prove this lemma since it is easier than the previous one and requires no new ideas. The two previous lemmata state, in effect, that no subwriting (by a reader or the writer) can advance more than two layers ahead the current subreading.

Lemma 4.3. Let r, i, $\overline{l_c}$, $t_{(5)}$, k and j be as in Lemma 4.1 (again, we do not assume that 'select' is invoked during r). Then there is a $k' \geq k-4$ such that reader j executes a subwriting on $A^{k'}[j,i]$ at an instant preceding $t_{(5)}$.

Proof. Intuitively, the lemma states that a subwriting by a reader cannot stay more than four layers behind the current subreading. At step (4) of r, $\overline{l_c}$ was obtained. Therefore, at an instant before $t_{(5)}$, the value of the subregister $L[j]$ must have been equal to l_j^c. So, there is an instant before $t_{(5)}$ that the local variable l of the reader j was given the value l_j^c-1. Let r_j be the high-level reading that executed this setting of l equal to the value of l_j^c-1. Consider the high-level reading r_j^- (i.e., the reading by j directly preceding r_j among the actions of reader j). Looking at the reader's protocol, and having in mind that during r_j, the local variable l was given the value of l_j^c-1, it is immediate to check that when r_j^- executes its subwritings on the j-th row, the variable l is $\geq l_j^c-3$. Therefore, there is a subwriting executed before $t_{(5)}$ on a subregister $A^{k'}[j,i]$, with $k' \geq l_j^c-3$. Since $k = l_j^c$ or $k = l_j^c+1$, the lemma follows.□

Lemma 4.4. Suppose that r and i are as before and suppose that at an instant t, r executes a subreading on $A^k[0,i]$. Then the writer at an instant $<t$ executes a subwriting on a subregister $A^{k'}[0,i]$, with $k' \geq k-1$.

Again, the above lemma is the counterpart of Lemma 4.3, when the second processor involved is the writer rather than the reader. We do not give a formal proof, since no new ideas are involved. By the two previous lemmata, we have that a subwriting (by a reader or the writer) cannot stay more than four layers behind the current subreading.

Consider now a run of the finite-layer protocol. Assume that during this run, for all variables with a layer-number, apart from their value modulo 7, also their actual value (i.e., the value they would have if all arithmetic operations were carried out in the set of integers) is considered. The comparisons on these variables that we do below, refer to this actual value. Intuitively, we consider a Riemann surface of the layers visited during the run of the finite case. In other words, intuitively again, instead of thinking of the layers as wrapping onto themselves and thus, close a cycle, we think of them as advancing to a higher level forming a helix. Then, notions like 'at a layer behind', 'at a layer ahead' and all comparisons between layer-numbers refer to their actual value, i.e., to the Riemann surface. Observe that, apart from the clearing ahead, on the Riemann surface, we actually have the protocol for the infinite case. The consideration of the actual values of layer-numbers is only done for the sake of formulating a proof. Naturally, we do not assume that these values are in any way considered by the processors.

To show that problem (a) mentioned at the beginning of this section cannot occur, we proceed as follows: Suppose that at an instant t, processor i ($i \geq 1$), during a high-level reading r, executes a subreading on $A^k[j,i]$ ($j \geq 0$). If this subreading is useful (i.e., if either $j=0$ or, otherwise, procedure 'select' is invoked during r), then by Lemmata 4.1 and 4.2, every subwriting preceding this subreading will take place on a subregister $A^{k'}[j,i]$, with $k' \leq k+2$. So, having ≥ 3 layers, we guarantee that if $k' > k$, then $k' \neq k$ modulo the number of layers. But this shows that problem (a) cannot occur.

We proceed now to problem (b). Assume that t, i, j and k are as before. By Lemmata 4.3 and 4.4, we have that before t, a subwriting W will be executed by processor j, at a subregister $A^{k'}[j,i]$, with $k'+4 \geq k$. So, if before the execution of W, 4 layers ahead are cleared, problem (b) cannot occur. We have to show though that this clearing ahead does not interfere with subreadings which will take place after the starting of the clearing and which will be executed at layers behind k'.

Towards a proof of that, assume that at an instant t_e, a printing of E is executed by j on $A^{k'+q}[j,i]$ $(1 \le q \le 4)$. Assume that this printing is part of the clearing ahead associated with subwriting W. Assume also that a subreading by i is executed on $A^m[j,i]$, at an instant $>t_e$, at a layer $m \le k'$. We have to show that $m \ne k'+q$ modulo the number of layers. Examining the protocols, we see that after the clearing ahead associated with W had started, and before W has been executed, there is no subaction by j other than printing of E's. But these printings pertain only to the finite case. So, in the infinite case, i.e., in the run viewed as taking place on the Riemann surface, all subactions of j preceding W come before t_e. Therefore, they precede the subreading on $A^m[j,i]$. So, the Lemmata 4.1 and 4.2 can be applied to obtain that $m \ge k'-2$. Therefore, $m \ge (k'+q)-6$. Since now $m \le k' < k'+q$, we have that $m \ne k'+q$ (mod 7). That means that with seven layers, the clearing ahead does not interfere with future subreadings at layers behind. This ends the correctness proof of the finite-layer protocol.

3.5. Conclusions.

Since in the previous section, we have proved that seven layers suffice, it is immediate to check that our implementation of a 1WnRbB register requires $O(n^2)$ 1W1R$O(n+b)$B atomic registers. Combining this result either with our four-tracks protocol or with Lamport's (1986) result, we get that

Theorem. A 1-writer, n-reader atomic register with b bits can be implemented with $O(n^3+n^2b)$ safe, 1-writer, 1-reader, boolean (1-bit) registers.

Acknowledgements.

Many thanks to Baruch Awerbuch for participating in the early stages of this work and Lambert Meertens for numerous useful conversations. Yoram Moses and Kostas Oikonomou supplied some pointers to the literature on metastable operation.

Burns and Peterson (1987) commented on a preliminary version of this paper, and refer to a construction of their own for an atomic, multireader register from atomic, 1-reader registers.

References.

Awerbuch, B., L.M. Kirousis, E. Kranakis and P.M.B. Vitányi (February 1987), "On Proving Register Atomicity," Technical Report CS-R8707, Centrum voor Wiskunde en Informatica, Amsterdam.

Bloom, B. (June 1986), "Constructing Two-writer Atomic Registers," Manuscript, Massachusetts Institute of Technology.

Burns, J.E. and G.L. Peterson (January 1987), "Comments on: Atomic Multireader Register (Detailed Abstract)," Technical Report GIT-ICS-87/08, Georgia Institute of Technology.

Chapiro, D.M. (1984), "Globally-asynchronous Locally-synchronous Systems," Doctoral Dissertation, Stanford University, California.

Kirousis, L.M., E. Kranakis and P.M.B. Vitányi (January 1987), "Atomic Multireader Register (Detailed Abstract)," Technical Report CS-8704, Centrum voor Wiskunde en Informatica, Amsterdam.

Lamport, L. (1986), On interprocess communication. Part I: Basic formalism. Part II: Algorithms, *Distributed Computing* 1, pp. 77-101.

Marino, L.R. (1981), General theory of metastable operation, *IEEE Transactions on Computers* C-30, pp. 107-115.

Misra, J. (1986), Axioms for memory access in asynchronous hardware systems, *ACM Transactions on Programming Languages and Systems* 8, pp. 142-153.

Papadimitriou, C.H. (1979), The serializability of concurrent database updates, *Journal of the ACM* 26, pp. 631-653.

Peterson, G.L. (1983), Concurrent reading while writing, *ACM Transactions on Programming Languages and Systems* 5, pp. 46-55.

Peterson, G.L. and J.E. Burns (December 1986), ''Concurrent Reading While Writing II: The Multi-writer Case (Preliminary Report),'' Technical Report GIT-ICS-86/26, Georgia Institute of Technology.

Singh, A.K., J.H. Anderson and M.G. Gouda (December 1986), ''The Elusive Atomic Register Revisited,'' Technical Report 86.30, University of Texas at Austin.

Vitányi, P.M.B. and B. Awerbuch (1986), Atomic shared register access by asynchronous hardware, *in* ''Proc. 27th IEEE Annual Symposium on Foundations of Computer Science,'' pp. 233-243.