Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

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

Atomic multireader register (detailed abstract)

Computer Science/Department of Algorithmics & Architecture Report CS-R8704 January

The Centre for Mathematics and Computer Science is a research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a nonprofit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

Atomic Multireader Register
(Detailed Abstract)

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)

We present: (i) A new implementation of an atomic, 1-writer, 1-reader, m-valued register from $O(\log m)$ safe, boolean registers (i.e., from scratch). The solution uses neither copying (of the values to be written) nor repeated reading. (ii) An implementation of an atomic, 1-writer, n-reader, multivalued register from $O(n^2)$ atomic, 1-writer, 1-reader, multivalued registers. Both constructions rely on the same idea. In a sense (ii) is a generalization of (i). This closes the last gap in the atomic shared register area. Together with some earlier constructions these results show how to construct atomic, multireader, multiwriter registers from - basically - elementary hardware like flip-flops.

1980 Mathematics Subject Classification: 68C05, 68C25, 68A05, 68B20.
CR Categories: B.3.2, B.4.3, D.4.1, D.4.4.
Keywords and Phrases: Register, run, safe, regular, atomic, boolean, flip-flop, reader, writer.

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

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 Defense Advanced Research Projects Agency (DARPA) under Contract N00014-83-K-0125.

Report CS-R8704
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

locking. Thus, although it *seems* that the actions are executed concurrently, deep in the system they are *actually* executed serially in some order. It has been pointed out in [Lamport1986a] 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 synchronisation 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 methods to control the communication between asynchronous participants, which may make such solutions 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. The purpose of the present investigation is to examine the feasibility of such general interfaces.

In particular, we analyse 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 either about the relative durations of the reads and writes or about the actual timing of the lower level constituent operation executions.

1.1. Definitions

A *register* is a black box with read and write terminal(s). To each terminal we can attach a single processor. One processor can be attached to several terminals. A processor attached to a write terminal can execute a write operation by following a protocol, which executes a sequence of operations through the write terminal. A processor attached to a read terminal can similarly execute a read operation. We assume that there is a *precedence* relation among the operations (which is a strict partial ordering i.e., irreflexive and transitive). Operations incomparable under this precedence relation are considered to be *concurrent.* Here, we restrict ourselves to 1-writer registers, i.e., there is but one write terminal. We assume that the write operations are totally ordered by the precedence relation. A write operation writes a value from a given domain, i.e., 0 or 1 for a boolean, and a value between 0 and $m-1$ for an m-valued register. A read operation returns a value written by a write.* We distinguish the following three types of registers in order of increasing strength [Lamport1986a]:

1. A register is *safe* if a read that has no concurrent writes returns the value written by the last write that precedes it.
2. A register is *regular* if any read returns either the value written by a write that is concurrent with it or the value written by the last write that precedes it.
3. A register is *atomic* if it is regular and if it cannot happen that the two writes, which wrote

* It is convenient (although not necessary) to assume the existence of a global (i.e., refering to all operators) time-reference system. Under this assumption, to each operation there corresponds a time interval within the bounds of which the operation is executed. An operation then precedes another if the finish time of the first is before the start time of the second.

the values returned by two reads, are ordered in the opposite direction than their corresponding reads.

We allow only *passive* serialization of operations. In a correct implementation no operation ever *waits* for any other operation. More precisely, let a register *implementation* consist of a set of subregisters (which are simpler in some way) such that the read- and write terminals of the subregisters are attached to the appropriate processors, together with a protocol these processors excecute. For each read or write of the register, this protocol specifies a sequence of reads and writes to the subregisters and some intermediate computation steps of the executing processor. The implementation has *no waiting* if all processors executing an operation always make progress (execute a next step) and there is a finite upper bound, expressed in terms of the parameters of the register implementation, which bounds the number of suboperations of a read or write as follows.

1. An operator executing a read or write to the register cannot be forced to execute a number of reads and writes to the subregisters, which exceeds this bound, by an adversary which can completely schedule all the (sub)operations of all other processes.
2. An operator which starts a read or write to the register always terminates.

No waiting implies *no livelock*. Livelock exists when an operator always makes progress, but is prevented from terminating by the other operators. No waiting also implies *no deadlock* and *no lockout* (an operator cannot be prevented from making progress).

In contrast with concurrency control where we actively serialize concurrent actions, in an atomic register the operations are *actually* executed concurrently, interleaved at lower levels, while it only *seems* as if they are executed serially.

2. History of the Problem and Results of the Paper

To simplify the notation from now on and for the rest of the paper the expression n-writer, m-reader, k-valued register will be abbreviated as nWmRkV register.

Atomic 1WnRmV registers have been constructed by Peterson [Peterson1983a] using 2 safe 1WnRmV registers, n safe 1W1RmV registers, $2n$ atomic 1W1R2V registers, and 2 atomic 1WnR2V registers. For n=1, an atomic 1W1RmV register can be implemented with 3 safe 1W1RmV and 4 atomic 1W1R2V registers. Lamport [Lamport1986a], noting that current applicable off-the-shelf basic hardware is of the 1W1R2V variety (i.e., flip-flops), implements atomic 1W1RmV registers from safe 1W1R2V registers. Hence, this reduces to Peterson's solution, apart from getting rid of the remaining 4 atomic 1W1R2V registers. In [Vitányi1986a] a construction is presented of atomic nWnRmV registers from atomic 1W1R∞V registers. Secondly, a (preliminary) construction of atomic nWnRmV registers from atomic 1WnR($f(m,n)$)V registers, with $f(m,n)<\infty$ for $m,n<\infty$, is given.

The *central problem* though, had remained unresolved. Namely, to construct atomic 1WnRmV registers from atomic 1W1R($f(m,n)$)V registers, with $f(m,n)<\infty$ for $m,n<\infty$. Now the closest we can come mathematically to a physical flip-flop is a safe 1W1R2V register. Therefore, what we ought to *really* aim at is a construction of atomic 1WnRmV registers from safe 1W1R2V registers. We provide such a construction below. One nice thing is that the same central idea is used both for the implementation of the atomic 1W1RmV register from safe 1W1R2V registers, as well as for the implementation of atomic 1WnRmV registers from atomic 1W1RkV registers. This provides a comprehensive approach for building the most general single writer register from "basic hardware." In particular, we present:

1. A new implementation of an atomic 1W1RmV register from $O(\log m)$ safe 1W1R2V

registers (i.e., from scratch). The solution uses neither copying of values to be written, nor repeated reading of values as in [Peterson1983a, Lamport1986a].

2. An implementation of an atomic 1WnRmV register from $O(n^2)$ (less than $5n^2$) atomic 1W1R($5 \cdot m \cdot 4^n$)V registers (e.g., constructed as in 1)

Both constructions rely on the same idea. In a sense, 2 is a generalization of 1. This closes the last gap in the atomic shared register area. Together with some earlier constructions, this shows how to construct atomic multireader multiwriter registers from "elementary hardware like flip-flops" (i.e., safe 1W1R2V registers).*

3. The 4-Tracks Paradigm

The basic idea used in the protocols to be described below originated from an attempt to construct a protocol that would implement read and write operations on *four tracks*. Consider four tracks (numbered 1,2,3,4) each consisting of an array of (an equal number) 1W1R2V registers (Figure 1).

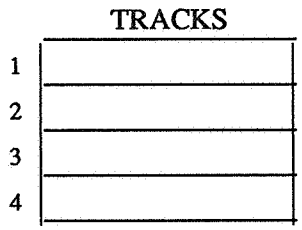

Figure 1: The 4 tracks

Two operators, a writer and a reader, each want to write and read respectively on the four tracks. A write of a number k on a track consists of writing the binary representation of k on the registers of this track (one bit per register). Similarly, a read from a track consists of reading a value (written in binary) on the registers of the track. It is desired to design a protocol which will guarantee that the following conditions are satisfied:

1. the operators are allowed to run concurrently and asychronously without any operator ever being locked out or even being forced to wait (*lockout-free* and *wait-free*),
2. while an operator is executing its job on a track the other operator is not allowed to enter the same track (*collision-free*),
3. the writer after finishing its job on a track moves to a new track, leaving its old track free for the reader to read (*writer-track-disengagement*),
4. if the reader is busy reading on a track then the writer writes back and forth between two free tracks (*writer-alternation*), until the reader finishes its reading,
5. if the writer is busy writing on a track then the reader must keep reading the same track over

* Let us stress here that safe 1W1R2V registers are mathematical concepts, while flip-flops are physical objects. Bistable devices like flip-flops are prone to *metastable* operation, see e.g. [Marino1981a, Chapiro1984a]. If the input that causes the bistable to change state is marginal, it may leave the bistable in a metastable state or metastable region that is between 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.

and over again until the writer finishes its job in which case it will move into the track that was just completed by the writer (in order to make sure that it reads the last write) (*writer-chasing*).

It should be clear that under such a protocol only the writer should be allowed to alternate between two tracks not currently used by the reader. This is natural because that way it is guaranteed that the writer always leaves its previous version intact for the reader to read. However, it would be wrong to allow the reader to do the same, because when the writer is busy working on a track the reads performed by the reader will alternate between a new and an old write, thus contradicting the atomicity requirement for registers. Moreover, the reader should always be *chasing* the writer in order to make sure that the *latest written version* is read.

In the next section we will describe how to implement the 4-track protocol using a switch consisting of constant number of safe 1W1R2V registers. This constant is independent of the length of the tracks. This is actually the only problem we need to solve. Namely, the switch ensures that no two operators can be engaged on the same track concurrently. Therefore, it is obvious that safe 1W1R2V registers suffice to implement the tracks in this algorithm. Since $\log m$ bits (consecutively written on the registers of a track) are enough to represent any number from 0 to $m-1$, the four-track protocol guarantees the implementation of an atomic 1W1RmV register by $O(\log m)$ boolean, safe registers.

4. Implementing 1-writer, 1-reader, atomic registers

The following protocol works on four tracks numbered 1,2,3,4. The machine parts consist of the four tracks above and an infinite sequence of *twin* registers (RQ_t, A_t), $t = 1,2,\cdots$. Later on it will be shown how to get rid of the infinitely many registers. Each of the registers RQ_t, A_t are of the 1W1RcV type, with c a fixed constant.

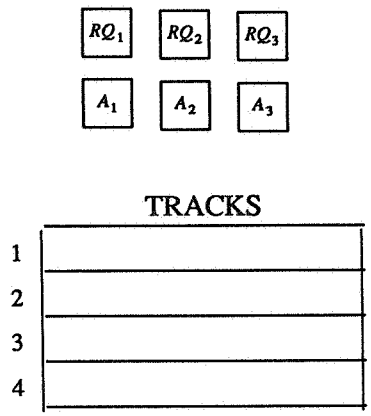

Figure 2: The 4-tracks register

The top line of registers consists of the *reader request registers*, denoted RQ_t, while the bottom line of registers consists of the *writer registers*, denoted A_t. The registers RQ_t assume the values E(mpty) and P(lease), while the registers A_t assume the values E(mpty), 1,2,3,4 (that correspond to track-numbers). In addition, registers RQ_t (respectively A_t) are read by the writer (respectively the reader) and written by the reader (respectively the writer). Each operator is endowed with local variables, in the following way.

1. There is a local variable t, one for each operator, assuming integer values and which indicates on what register an action is to be taken.
2. The writer has a local variable called *vb* (*verboden*), which assumes as values subsets of $\{1,2,3,4\}$ of size at most 2. The elements of *vb* indicate at what tracks the writer should *not* go.
3. The writer has a local variable m, which assumes the values 1,2,3,4 and specifies the track number on which the writer chooses to write. After the completion of writing on this track, the value of m is printed on the appropriate register for the reader to see. Also, there is a variable m_{old} which at any instant assumes the previous value of m. Initially, m has no value, t is 1, *vb* is empty, and the registers RQ_t and A_t are all (E)mpty.

The protocol can now be described as follows:

Writer writes value v:
1. read RQ_t;
2. **if** $RQ_t = P$ **then** set $t := t+1$ and $vb := \{m_{old}, m\}$ (the set consisting of the numbers of the last two completed tracks);
3. $m_{old} := m$; $m := x$, $x \notin \{m\} \cup vb$; write v on track m;
4. write m on A_t.

Reader:
1. read A_t;
2. **if** $A_t = E$ **then** ($t := t-1$; read A_t) **else** (write P on RQ_t; read A_t);
3. read value from track m (m the number obtained in step 2);
4. $t := t+1$.

If the writer sees a P on RQ_t it prints the track-number it has just completed onto register A_{t+1}. Clearly, there are at most two different track-numbers that can appear on A_t at any instant that follows the printing of a P on RQ_t. The first one is the track-number that was there before and after the printing of that P. The second one is the track-number that might be printed after the printing of P in RQ_t, while P was not yet printed when RQ_t was checked in the write involved. Obviously, there can be at most one such write. These two numbers are put by the writer into its *verboden* set. This set is not changed until the writer sees a new P, an indication that the reader is "on the move" again.

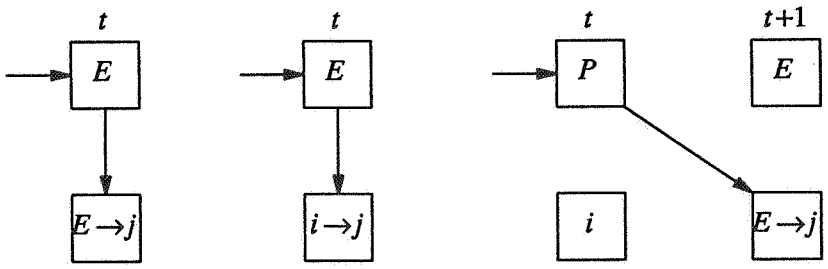

Figure 3: The three possible actions of the writer

In figure 3, the writer is at stage t and one of the following situations can arise. Either it reads $RQ_t = E$, in which case it changes the contents of A_t to the value determined by the track number j just completed (recall that A_t contained either the value E(mpty) or the value i of the track

number previously completed), or else it reads $RQ_t = P$, in which case it moves to stage $t+1$ and sets $A_{t+1} = j$ (j is the number of the most recently completed track), leaving the track number i (it wrote before) on A_t for the reader to read.

After the reader has printed a P on a register RQ_t, it reads the register A_t and moves onto the track whose number is indicated there. In fact, the reader goes to a track whose number is one of the two mentioned in vb. To guarantee that the reader does not move onto a register RQ_t with nothing to read on the corresponding A_t, we make sure that the reader prints a P only after it has checked that there is something to be read on A_t. If there is not, it backs up and reads A_{t-1}. In this case it does not print a P, since there is already one. Also, the value obtained from A_{t-1} is acceptable, since when the reading started there was nothing on A_t.

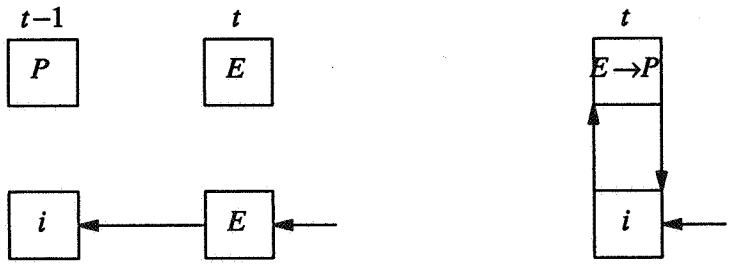

Figure 4: The two possible actions of the reader

In figure 4, the reader is at stage t and one of the following two situations can arise. Either it reads $A_t = E$, in which case it moves back and reads the track value printed on A_{t-1}, or else it reads $A_t \neq E$, in which case it sets $RQ_t = P$ and reads the value printed in A_t (which may very well contain a different value than the one he read before, due to a write action by the writer).

Clearly, the registers on the tracks need only be safe, since the tracks are collission-free. The (boolean) registers RQ_t need only be regular (and hence safe). This is so because a writer will never reread such a register, once it gets the value P. The registers A_t can be assumed to be regular. Indeed, the reader may move back onto A_t for a second time. But the writer reading a P in RQ_t will write on A_{t+1}. Hence, there are at most two possible numbers that may be read on A_t, and those are written by consecutive writes. Namely, writes such that the last one starts while RQ_t contains E and ends while RQ_t contains P, and the write immediately preceding. So, all we have to guarantee is that if the reader sees a new value the second time it reads a register then it picks the latest one among the two consecutive values it has seen. To accomplish this we make the writer write not only the value a, but also a *serial* number s. At each new write of a value b it sets $s := (s+1) \bmod 3$. For example, the following is a legal sequence of six writes:

$$(a,0), (b,1), (c,2), (d,0), (e,1), (f,2).$$

It is then clear that given any two successive writes the reader can decide which one is the last. Notice though, that now the registers A_t are regular and 15-valued.

Finally, it remains to show how to get rid of the infinite sequence of registers RQ_t and A_t. Since the reader only advances if the writer has advanced, and vice versa, we can replace the infinite sequence of registers by a sequence of c registers which are cyclically used, with c a small constant like 3. Hence, the values of t in the algorithm need only be reduced $\bmod c$. We then need to precede the write subaction of each write with a step which sets $A_{t+1} := E$, and the write subaction of each read with a step which sets $RQ_{t+1} := E$. Counting the needed subregisters we conclude that we can implement an atomic 1W1RmV register from $4 \log m$ safe 1W1R2V

registers (4 tracks) plus 3 safe 1W1R2V registers (3 RQ-registers) and 3 regular 1W1R15V registers (3 A-registers).

Using Lamport's [Lamport1986a] constructions 3 and 4, it follows that we can implement regular 1W1R15V registers by 14 safe 1W1R2V registers. To sum up it has been shown that

Theorem. *We can construct an atomic, 1-writer, 1-reader, m-valued register from $45 + 4\log m$ safe, 1-writer, 1-reader, boolean registers.*

5. Implementing a 1-writer, n-reader Register

In this section we generalize the previous construction in order to obtain an implementation of an atomic 1WnRmV register.

5.1. Architecture.

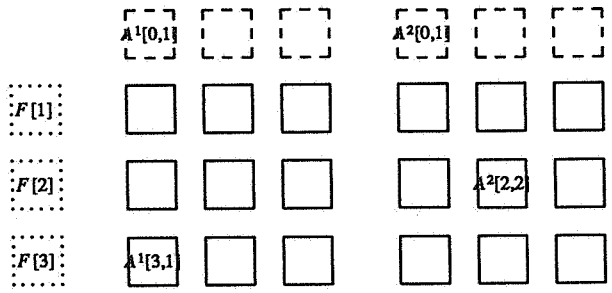

Figure 5: Two Matrices of the atomic, 1-writer, 3-reader register

Each box pictured is an atomic 1W1RkV register (e.g., as constructed in the previous section). There is a column $F = (F[1], \ldots, F[n])$ of n flag registers (dotted boxes) and an infinite sequence $(A^t)_{t>0}$ of $(n+1)\times n$ matrices. Later we show that 4 matrices suffice. The row $(A[0,1], \ldots, A[0,n])$ (dashed boxes) is written by the writer, while row $(A[i,1], \ldots, A[i,n])$ (solid boxes) is written by reader i. In addition, the j-th reader can read the entries of the j-th column $(A[1,j], \ldots, A[n,j])$. All registers in F can be read by the writer, and $F[i]$ can be written by reader i. (We can consider F as the 0th column of all A^t's.)

5.2. Algorithm

The writer maintains an array $(t_1, \ldots, t_n)$ of local variables ranging over the positive integers and a variable s taking values 0,1,2,3,4. The writer writes these variables in registers which can be read by the readers. When the writer writes any value v to the conceptual shared register, it writes $(v, (t_1, \ldots, t_n), s)$ as value to the subregisters. s is a local variable of the writer which is incremented by 1(mod 5) at each write. Each reader maintains a local variable (*flag*) taking values 0,1, and writes it in a register (the ith entry $F[i]$ of F, for reader i) which can be read by the writer. In addition, each reader has a local variable t. Reader i's variable t corresponds to the writer's variable t_i the same way the reader's t correspond to the writer's variable t in the algorithm of the previous section.

Writer writes v_{new}:
Writer has finished the writing of $(v_{old}, (t_1, \ldots, t_n), s)$ and wants to write the new value v_{new}.

1. Check the flag column F and notice the changes that have taken place in each entry, i.e., flag.
2. for all i in $\{1,...,n\}$

   if there is a change in the ith flag $F[i]$, then set $t_i := t_i+1$;

3. $s := (s+1) \bmod 5$;
4. for all i in $\{1,...,n\}$

   write $W=((t_1,...,t_n),s)$, as well as v_{new} in $A^{t_i}[0,i]$.

Reader i reads:

1. read $A^t[0,i]$;
2. if $A^t[0,i]=E$ then($t := t-1$; read $A^t[0,i]$) else ($F[i] := \neg F[i]$; read $A^t[0,i]$);
3. for all k in $\{1,...,i-1,i+1,...,n\}$

   read $A^{t_k}[k,i]$ and $A^{t_k+1}[k,i]$;

4. read $A^{t+1}[0,i]$ (possibly for the second time);
5. if $A^{t+1}[0,i] \neq E$ (i.e., write finished *after* this read began) then read value from $A^t[0,i]$ else call procedure *select*;

Procedure *select*:

(i) determine, among items obtained in step 3, the ones that have $t_i = t$;

(ii) if there is one with its s equal to $s_0+1 \pmod 5$ (s_0 the s-value in $A^t[0,i]$ as read in step 2) then return its associated value v else return value v associated with $A^t[0,i]$.

6. for all k in $\{1,...,i-1,i+1,...,n\}$

   write the selected value and associated $W=((t_1,...,t_n),s)$ in $A^t[i,k]$.

7. $t := t+1$.

5.3. Explanation

The *writer* writes on different layers "matrices" for each reader. If the writer sees that the flag register of reader i has changed, then it advances one layer with respect to reader i so as not to interfere with a concurrent read of the reader. The writer *stamps* each new value with a sequence number $s \pmod 5$.

Reader i starts by reading the *next* layer (t) after the one it used in the last read. If the writer has not written there yet then the reader backs up to the previous layer ($t-1$). One way or the other, it obtains the (writer-written) contents of a subregister it shares with the writer. Call the selected layer l. (Note that while the reader is backing up the writer may suddenly write in the layer (t) the reader just left.) If the reader does not back up then it changes its entry in the flag array F to signal the writer to go to the next layer ($t+1$). (Each write which starts after this will go to a layer $t'>t$.)

Subsequently, for all $k \neq i$, reader i reads the contents of the subregister ($A^{t_k}[k,i]$) it shares with reader k (in the layer t_k), as well as the contents of the subregister $A^{t_k+1}[k,i]$ (in the next layer). Now the reader reads the subregister it shares with the writer in layer $l+1$. If the reader backed up at the beginning, then this is the subregister it backed up from.

Case 1. If this subregister is now nonempty then there has been a *first* write, say w, ending after the current read began, such that this write wrote in this subregister. This means that the *last* write before w, say w', was written in the subregister of layer l from which the reader already obtained the writer-written contents. Moreover, the value written by w' persisted during this read.

Therefore, it is a valid value to return. Hence the reader reads the latter subregister again, and returns the value v it contains.

Case 2. If this subregister is still empty, then the reader chooses between the contents of the writer's subregister it originally read and the contents of the subregisters it shares with the other readers (and has read). Then it *selects* as follows. It considers only the contents resulting from writes which wrote to its own current layer, i.e., such that $t_i = l$ in the array $(t_1, \ldots, t_n)$ written in that subregister. By assumption, in Case 2 no write has written in a subregister the reader i shares with the writer during the current read. So there is no write which starts after this read began and ends before this read ends. Therefore, the values in the subregisters shared with the other readers (written with $t_i = l$) can be at most two behind or one ahead. Hence attaching s to the written value and incrementing it mod 5 suffices to select the latest one among the values obtained from $A^{t_k}[k,i]$ with respect to the value from $A^l[0,i]$. The reader returns the associated value v. Finally, the reader writes the values it selected to the subregisters it shares with the other readers in the layers indicated by W, and increments t by 1.

5.4. Finitely Many Layers

Since both the reader and the writer always write adjacent layers we can cycle through a bounded number of them. We actually need only a small constant number c of them ($c \leq 4$). In the algorithm the t's are to be reduced mod c. Moreover, each write subaction by the writer is preceded by a step

for all i in $\{1, \ldots, n\}$

$A^{t_i+1}[0,i] := E$

and each write subaction (on the main matrices) by a reader i is preceded by a step

for all k in $\{1,\ldots,i-1,i+1,\ldots,n\}$

$A^{t+1}[i,k] := E$

Caution: In the case of finitely many layers the following problem may arise. Cell $A^{t_k}[k,i]$, which is polled in step 3 of reader i's algorithm, may contain a value which appeared after the value it had when t_k was determined in step 2. Namely, if the kth reader and the writer alternated some reads and writes (at least c times). Fortunately, in that case the value in cell $A^{t_k}[k,i]$ is not used in step 5 at all!

Combining this with the construction of an atomic 1-writer, 1-reader register desribed in the previous section we obtain the following theorem. (Note that each matrix entry needs to store s the vector W and a value v.)

Theorem. *We can construct an atomic, 1-writer, n-reader, m-valued register using* $O(n^2(n+\log m))$ *safe, 1-reader, 1-writer, boolean registers.*

Remark: Obviously, we can also store the contents of each $(A^1[i,j], \ldots, A^4[i,j])$ as an array in a single entry $A[i,j]$ of a single $(n+1)\times n$ matrix A. That way we implement an atomic, 1-writer, n-reader register with about $(n+1)^2$ atomic, 1-writer, 1-reader registers. Expressed in terms of the elementary, safe, 1-writer, 1-reader, boolean registers this amounts to the same thing as before.

6. Multireader Multiwriter Register

Splicing the above onto the results of [Vitányi1986a] we obtain:

Theorem. *We can construct an atomic, n-writer, n-reader, m-valued register from* $O(n^3(n^2 \log n + n \log m))$ *safe, 1-reader, 1-writer, boolean registers.*

Acknowledgement

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.

References

Chapiro1984a. Chapiro, D.M., "Globally-asynchronous locally-synchronous systems," Tech. Rept., Stanford University, Department of Computer Science (October, 1984).

Lamport1986a. Lamport, L., "On interprocess communication, Parts I and II," *Distributed Computing* **1**, pp. 77-85, 86-101 (1986).

Marino1981a. Marino, L.R., "General theory of metastable operation," *IEEE Transactions on Computers* **C-30**, pp. 107-115 (1981).

Peterson1983a. Peterson, G.L., "Concurrent reading while writing," *ACM Transactions on Programming Languages and Systems* **5**, pp. 46-55 (1983).

Vitányi1986a. Vitányi, P.M.B. and B. Awerbuch, "Atomic shared register access by asynchronous hardware," in *Proceedings 27th Annual IEEE Symposium on Foundations of Computer Science* (1986).