A Conditional, Interlock-free Store Instruction*

Daniel P. Friedman

David S. Wise

Computer Science Department

Indiana University

Bloomington, Indiana 47405

TECHNICAL REPORT NO. 74

A CONDITIONAL,
INTERLOCK-FREE STORE INSTRUCTION

DANIEL P. FRIEDMAN
DAVID S. WISE

REVISED AUGUST, 1980

*Research reported herein was supported (in part) by the National Science Foundation under grants numbered MCS75-06678 A01 and MCS77-22325.

A preliminary version of this paper was presented at the 16th Allerton Conference on Communication, Control, and Computing.

A CONDITIONAL, INTERLOCK-FREE STORE INSTRUCTION*

Daniel P. Friedman

David S. Wise

Abstract:

The sting store instruction generalizes the imperative that alters the contents of a memory location. It allows qualification on that instruction to localize it to a single field within a memory word and to make it conditional on the extent value of a single bit in that word. The field selection and the conditional test are intended to be made remotely from the processor, as local to the memory word as possible.

This generalization is particularly attractive in multiprocessing applications where memory is shared and synchronyzation of memory altering operations is necessary. The sting instruction, dispatched over a virtual circuit, is shown to be sufficient by itself to solve useful problems, without processor/processor synchronization implicit in semaphore-like schemes. A simple example -- a child's Easter egg hunt -- is presented in detail and proven correct, a practical application is outlined, and a possible memory circuit is included.

Key words and phrases: memory, parallel processing, multiprocessing, synchronization, sting. CR categories: 4.32, 6.34.

Introduction

This paper introduces the sting store instruction as a primitive in a multiprocessing environment. It is intended as an expansion on the conventional "store" or "assign" primitive invented by von Neumann with a limited provision for signalling among processors sharing a memory "word" that an assignment has occurred.

There already exist several control primitives in the literature designed for the multiprocessing environment. These include semaphores [5], monitors [2], test-and-set [19], et al. All of these primitives offer the programmer some assurance of mutual exclusion, often necessary for multiprocessing algorithms (e.g. dual processor garbage collection [6, 12].) The need for mutual exclusion often arises in a programming style in which an individual processor makes repeated assignments to a single variable ("word" in the store) because any two processors may interleave their own assignments in a manner which interferes with each other. Since we are concerned with writing programs for unrestricted numbers of processors, every such exclusion adds an unforseeable overhead to run-time behavior. We are interested in a different programming style which has been less popular until recently and in which, therefore, there are fewer available multiprocessor "chestnut" programs. For the purposes of this paper, we characterize such a style as the single-assignment approach of Tessler and Enea [21], although applicative or functional programming [1, 3, 13, 16, 17] is a more familiar (and motivating) "sub-style" when such a style is extended from a single processor to a multiprocessor environment, several processors might each be

attempting to store (perhaps different) values only once in common variables. A control mechanism is necessary to ensure that, in fact, only one assignment occurs in the shared memory, even though several have been dispatched from the several processors. Although easily implemented with mutual-exclusion, the control required is considerably more simple as we shall see, than generalized mutual-exclusion, with the result that the multiprocessing programs that arise under such a style are easier to create and prove and ought to be more easily implemented and maintained under a higher degree of parallelism. We expect this result since there is less of a requirement to establish freedom from interference [12] because the semantics of sting itself, provides some exclusion of assignment, rather than requiring an appeal to the stronger sense of mutual exclusion of processes.

In order to characterize the memory architecture of multi-processors, we shall appeal to a message-passing concept from distributed processing [7]. Since we perceive the store (or main memory) as being separate from the processors (and perhaps "banked" in a manner to allow simultaneous accessing analogous to the simultaneous processing of the processors), let us perceive each processor and each memory word (or memory bank) to be nodes in a distributed processing network. We are not interested in the interconnection graph of this network, but we do require that every processor have a path to access every word of the main (global) memory (or store). The network acts as an asynchronous switch between processors and memories, which may

behave like a packet-switched or store-and-forward communication network. We do require, however, that communication between any given processor and any given word behave as a virtual circuit rather than datagrams [22]; that is, messages from any processor to any memory must be received within a fixed amount of time (bounded by an implementation constant) in precisely the order in which they were sent. One interconnection pattern which offers such behavior is the Banyan net [11] in which there is a unique path between any of the processors on one side of the net and any of the memory banks on the other. Because messages must be queued along this path, they necessarily arrive in the order of transmission; bounded time is guaranteed by alternating the binary choices at each active node of the network and by treating lost messages as system-wide errors. Such severe treatment of communication failures is to be expected in multiprocessing although it would be tolerated in distributed processing.)

The remainder of this paper is divided into four parts. The next section presents a syntax and semantics for the sting primitive based on PASCAL and the Floyd-Hoare scheme. In the following section we present and prove a simple example, the Easter egg hunt, useful perhaps as a system benchmarking test. Then we expand to a motivating example, the coaxing arbiter [10] in the next section. In the final section we compare the sting primitive to classic synchronizing primitives to justify its implementation and utility -- if not its power -- in a multiprocessor.

Syntax and Semantics

In this section we introduce the sting instruction into the above described multiprocessing architecture. This will be done by treating all of the common memory as records referenced only by pointers (addresses in the common sense). When simple variables are used in the following code, they will refer to registers local to the executing processor. In terms of conventional low-level programming then, shared memory is accessible only indirectly via a pointer resolution; this level of indirectness appears in place of the message passing of the distributed model above.

Fetching a value from common memory into a local register is denoted using PASCAL's up-arrow convention.

```
type  tipe = something; pointer = ↑tipe;

var  LOCALQ:  tipe;    Q:  pointer;
     {The values in both these variables may be in use
      by other processors.  Q references common memory.}

. .
    LOCALQ := Q↑
```

Using the distributed processing model, this statement would dispatch a request from this processor to Q's region of global memory, requesting that its contents be returned, to be stored in the local register LOCALQ. Processing might continue even before reception of Q's contents as long as LOCALQ were not involved; like the axiom following, this overlap of processing is thus possible only when LOCALQ has no other name involved in the overlap. The conventional Floyd-Hoare axiom for assignment [12] applies to this fetch when X has no other name involved (either Y or in P).

$$\{P^{X}_{Y\uparrow}\} \quad X:=Y\uparrow \quad \{P\}$$

(P^{x}_{y} represents the result of substituting y for every free occurrence of x in the predicate P.)

Where PASCAL would allow the store instruction

Q↑ := LOCALQ

we would use a special case of sting

sting Q with LOCALQ

to effect the storing of a new value in the common memory. Precluding common memory as the target of ordinary assignment statements (denoted by ALGOL's :=) has immediate advantages arising from the alias restriction in Floyd-Hoare semantics; it is quite difficult to meet that restriction when the left-hand-side is a variable-valued reference, because it is so easily aliased by another pointer or subscripted variable

A sting statement has the syntax

<sting statement> ::= <address><flag><field><value>

<address> ::= sting <expression>

<flag> ::= unless <identifier>|<empty>

<field> ::= in <identifier>|<empty>

<value> ::= with <expression>

The <expression> in the <address> portion must evaluate to a legal reference to global memory. A record must be at that address with subrecords (or fields) named by the <identifiers> in the optional <field> and <flag> portion if these are

nonempty; that is the <flag> portion must be of Boolean (logical) type. The <expression> in the <value> portion must evaluate to a value of the same type as a whole record when <field> is empty, or as the <field> subrecord otherwise. Only the <address> and <value> portions must be nonempty; absence of the <field> portion indicates that the store instruction affects the entire record and absence of the <flag> portion indicates that the store will occur unconditionally. The conditional store when <flag> is nonempty is the motivation for the generalized <sting> instruction, and we describe it below.

The motivation for the sting instruction is its conditional behavior when the <flag> phrase is nonempty; other instances are just conveniences of the general syntax. When a conditional behavior is invoked, say with the instruction sting ADDRESS unless FLAG in FIELD with VALUE the following steps occur without interruption:

```
if ADDRESS↑.FLAG then skip else ADDRESS↑.FIELD := VALUE
```

alternatively,

```
with ADDRESS↑ do if FLAG then skip else FIELD := VALUE.
```

This latter code is suggestive of our intention that the stinging processor not, itself, be involved in the conditional behavior. Using the distributed processing model of the previous section as a model, we imagine that the conditional behavior (nested within the with construct) occurs at the memory independently of the processor. After dispatching the FLAG, FIELD, and VALUE to the ADDRESS's memory, the processor is free to continue; if it should

subsequently refer to that word of common memory, the virtual circuit behavior of the memory switch guarantees that the sting store will have been reflected in that access. (See also add-to-memory [20] for similar behavior, only there the set of possible conditionals has been omitted.)

This behavior motivates the choice of the term, "sting," itself. The analogy with an insect's defense arises from the separation of the store instruction (via the virtual circuit) from its originator and from its dramatic and instantaneous reception at its target.

Operationally, the conditional behavior amounts to a read-conditional-write at the word in store, where the existing contents of the FLAG bit is gated onto the write-enable line. A general implementation of sting requires a wider path from processor to memory, since this store instruction not only specifies an ADDRESS and a VALUE, but also describes a FIELD (position and width at the ADDRESS) and the location of FLAG. In general this could considerably increase the size of a "store" message as it is sent to memory, but we expect that FIELD and FLAG will be restricted to a few allowable positions within a word in actual hardware (e.g. byte boundaries and sign bit, respectively) so that their descriptors would be substantially compressed. Indeed, the motivating applications [10], admit such a restriction.

The following four Floyd-Hoare style axioms summarize the semantics of the alternative forms of the sting instructions:

$$\left\{P^{\text{ADDRESS}\uparrow}_{\text{VALUE}}\right\}\ \text{sting ADDRESS with VALUE}\ \{P\}\ ;$$

$$\left\{(P\ \&\ \text{ADDRESS}\uparrow.\text{FLAG})\ \vee\ (P^{\text{ADDRESS}\uparrow}_{\text{VALUE}}\ \&\ \neg\ \text{ADDRESS}\uparrow.\text{FLAG})\right\}$$

$$\text{sting ADDRESS unless FLAG with VALUE}\ \{P\}\ ;$$

$$\left\{P^{\text{ADDRESS}\uparrow.\text{FIELD}}_{\text{VALUE}}\right\}\ \text{sting ADDRESS in FIELD with VALUE}\ \{P\}\ ;$$

$$\left\{(P\ \&\ \text{ADDRESS}\uparrow.\text{FLAG})\ \vee\ (P^{\text{ADDRESS}\uparrow.\text{FIELD}}_{\text{VALUE}}\ \&\ \neg\ \text{ADDRESS}\uparrow.\text{FLAG})\right\}$$

$$\text{sting ADDRESS unless FLAG in FIELD with VALUE}\ \{P\}\ .$$

Like other assignment axioms [12] these may not be consistent if the ADDRESS reference has an alias in the expressions for VALUE or P. Consistency is not threatened by an alias in FLAG or FIELD since these must be simple field identifiers, rather than expressions. An example of a proof using the second axiom is in the following section. It shows that the aliasing problem is considerably less than might be anticipated, because the invariants concerning common memory can be universally bound; universal quantification works because the uninterruptible conditional allows FLAGged consistency to be sustained across assignments to common memory.

With these conventions it is easy to distinguish fetches from common memory (by the "↑") from stores to common memory (using "sting") and local register manipulations (using neither).

A Detailed Example

In this section we present a detailed, although somewhat fanciful, example of the use of sting. This simple program is designed to be run simultaneously by an arbitrary number of processors over a common memory with no mutual exclusion. Sting instructions are the only synchronization tool. A proof using one of the axioms above is included, and a hardware implementation of the necessary shared memory has already appeared [9].

An Easter egg hunt is a game played by millions of children every year. Before such an event many dyed eggs are hidden on a playing field. The hunt begins on a public signal, whereupon many children scurry about the field, each claiming as many eggs as each can find. Of course eggs can't be shared, so some may not find any. Strategy is significant; an inexperienced player often follows a more experienced player about the field, wondering why he never finds unclaimed eggs in the same spots that the experienced player just looked. If all players follow independent search strategies (independent of the hiding strategy also) then the uniformly hidden eggs will be equitably distributed at the end of the game.

Our simulation of the Easter egg hunt uses a common memory of MEMSIZE words with addresses ranging from 1 up to MEMSIZE inclusive. (It will be necessarily to set MEMSIZE as a large prime). Each memory location represents an egg and contains

all (at least relatively) prime.

Corollary: For all j, $n_{j,MEMSIZE} = 0$. The corollary yields a common termination condition for the generation above.

Let us assume initially that I↑.CLAIMED = false for all 0<I<MEMSIZE, indicating that no eggs have been claimed and first satisfying the "claimed" invariant,

C = ∀I(0<I<MEMSIZE ⊃ (I↑.CLAIMED ⊃ 0<I↑.OWNER≤K)).

The strategy followed by the j^{th} child, using its local register I, is then

```
begin
    I := p_j;
    while I ≠ 0 do
    begin
        sting I unless CLAIMED with (true, j);
        I := (I + p_j) mod MEMSIZE
end end.
```

At this point we have justified the invention of this code fragment. It remains to establish its validity in a multi-processing environment. We shall do this by reformulating the code as a proof below, using ghost variables [18] (Also [15]) from the previous section, and the global claimed invariant, C, above. While the code is a trivial transformation of that above, we do introduce the subscript j on the index I and on the ghost variables to indicate that these values are local to one of the K active processors, and thus are protected from the activities of the others.

We introduce the sequencing invariant

$$S(j) = I_j = M_{j,GHOST_j} \;\&\; \forall i(0<i<GHOST_j \supset n_{j,i}\uparrow.CLAIMED).$$

The code is here expressed as one of many parallel programs (each identical but for the value of j and p_j) enclosed in **cobegin-coend** brackets, with the remark "Other K-1 children..." indicating the other identical replications of code. The order of the fragments inside these brackets is, of course, immaterial, but the indexing is meaningful for use in the inductive proof. The correctness of the multiprocessor code for an arbitrary number of processors is accomplished by an induction of K.

When K=1 (hence j = 1) this proof is a conventional Floyd-Hoare proof. The only interesting item is the pre- and post-conditions on the sting instruction maintaining Invariant C regardless of whether or not the contents at I_j actually changed. The second axiom above applies here.

We now consider the validity of this algorithm when there is an arbitrary number of children/processors. Since there is no mutual exclusion (i.e. the await statement [18]) there is no qualification necessary to the concept of interference-free [12]. We have a simple and intuitive concept of freedom from interference: no assignment or sting statement may interfere at all with the proof. The proof now proceeds (and expands) with an induction step (respectively, recursion) on K.

Assume that the proof of the algorithm is valid for K processors. Let us introduce a $K+1^{st}$ processor executing the same code into the cobegin-coend statement. This new processor has its own index--j, registers I_j and $GHOST_j$, and a prime number--p_j, none of which are in use by any other processor. (We note that a fixed value for MEMSIZE restricts the availability of smaller unused primes. Thus the induction is not validly extended to all integers, K, unless MEMSIZE grows with K, (say by advancing to the next prime with each increment to K.) No assignments to I_j or $GHOST_j$ interfere with any of the proofs of the other K processes, because these are private local registers. Similarly, the assignments and stings in the other K processes do not interfere with

```
begin  (*scorekeeper*)
    for  I := 1  to  MEMSIZE - 1  do  SCORE[I] := 0;
    for  I := 1  to  MEMSIZE - 1  do
        begin
            repeat  E := I↑  until  E.CLAIMED;
            SCORE[E.OWNER] := SCORE[E.OWNER] + 1
        end
end
```

It is quite possible that all scores are tallied before all children have quit hunting. Those that continue to hunt, of course, won't find any more eggs.

processor. This is called regression [10] and is benign since in the worst case the process proceeds at the pace of the slowest processor.

What we do not tolerate is regression from a "final state." Each process may be eventually coaxed into an immutable final state, whence no processor may duplicate or change the results in its record in common memory. In the single assignment (particularly, the applicative) languages which interest us, that final record is the value computed by that process and it would be wrong to allow any other processor to change it. In a lazy evaluation scheme one might associate a process with a suspension [8] and its final state with its ultimate value.

Using an EGG metaphor, we imagine that we have several eggs that we are trying to hatch. Each egg may or may not hatch, and we must hatch at least one. Hatching is encouraged by COAXing the contents of an egg; COAX is a function which returns slightly new contents of an egg to be stored in memory until later. Once an egg is hatched, however, it must not be touched. The problem is to signal a hatched egg, a final state, to all other processors which may be coaxing it, so that they do not touch it, even to store another hatched result.

Ordinary synchronization primitives can solve this problem, but they generally require some processor level synchronization to stop or start (block or free) other processors. The solution below uses sting to remove all synchronization to the memory, so that coaxing processors run with no inhibitions except access to

In fact the example in [10] actually had two such coaxable fields at each address, which required that the FIELD being stung be specified. That motivated the "in FIELD" option in the syntax and semantics set forth above, but not used here. (Those two fields correspond to the two fields in LISP operational semantics for cons.)

Comparisons and Conclusions

There are many synchronization primitives, defined operationally and formally in the literature. Some have been implemented directly in processor hardware. To our knowledge, this is the first which is proposed with the intention of being implemented exclusively in memory hardware of a multiprocessing system.

Controls like monitors [2], semaphores [5], test-and-set [19], or compare-and-swap [4] are described as inhibitions placed on the flow of control. A direct implementation would require much subliminal signalling among processors at every cycle in order that each would know that it could proceed. This communication is an undesirable burden if it is disjoint from other communication, because it constrains the growth of the system and because it increases the connections required to add a new processor into the system. Moreover, requirements for rapid signalling may constrain the number of possible signals to be much less than, say the size of the main memory. If such interprocessor communication is included in other communication streams (e.g. a bus), processors may be so overburdened with this important communication that they can make little progress.

Of course, one synchronizing primitive often can be used to emulate another, but we are not considering simulation-here. We are very much concerned with alleviating interprocessor synchronization to take advantage of the efficiencies of single-assignment or applicative-style programming. These approaches are already characterized by relative freedom from interference

among processors. Sting is an effort to extend this behavior into indeterminate programs [10].

In the case of sting the synchronization is handled over a virtual circuit between processor and shared memory and by a simple gating of one bit onto the write-enable line at the memory location. Figure 1 shows a schematic for a memory designed for the Easter egg hunt. The cost of the processor/memory switch will appear in any multiprocessor; we increase our return by designing it to help in synchronization. The cost of the gating is a slight decrease in the speed of common memory, which will never be noticed with many processors and significant switching delays.

Dedication: This paper is dedicated in memory of Douglas Goode, who developed the circuit presented here as Figure 1.

REFERENCES

1. Backus, J. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. Comm. ACM 21, 8 (Aug. 1978), 613-641.

2. Brinch-Hansen, P. Operating System Principles. Prentice-Hall, Englewood Cliffs, NJ, 1973.

3. Burge, W.H. Recursive Programming Techniques. Addison-Wesley, Reading, MA, 1975.

4. Case, R.P., and Padegs, A. Architecture of the IBM Systems/370. Comm. ACM 21, 1 (Jan. 1978), 73-96.

5. Dijkstra, E.W. Co-operating sequential processes. In Programming Languages, F. Genuys, Ed., Academic Press, London, 1968, pp. 43-112.

6. Dijkstra, E.W., Lamport, L., Martin, A.J., Scholten, C.S., and Steffens, E.F.M. On the fly garbage collection: an exercise in cooperation. Comm. ACM 21, 11 (Nov. 1978), 966-975.

7. Feldman, J.A. High level programming for distributed computing. Comm. ACM 22, 6 (June 1979), 353-368.

8. Friedman, D.P., and Wise, D.S. CONS should not evaluate its arguments. In Automata, Languages and Programming, S. Michaelson and R. Milner, Eds., Edinburgh University Press, Edinburgh, 1976, pp. 257-284.

9. Friedman, D.P., and Wise, D.S. Sting-unless: a conditional interlock-free store instruction. In Proc. 16th Allerton Conf. on Communication, Control, and Computing, M.P. Pursley and J.B. Cruz, Jr., Eds., University of Illinois, Urbana, 1978, pp. 578-584.

10. Friedman, D.P., and Wise, D.S. An approach to fair applicative multiprogramming. In Semantics of Concurrent Computation, G. Kahn, Ed., Springer, Berlin, 1979, pp. 203-226.

11. Goke, L.R., and Lipovski, G. J. Banyan networks for partitioning multiprocessor systems. In Proc. 1st Symp. on Computer Architecture, Computer Architecture News 2, 4 (Dec. 1973), 21-28.

12. Gries, D. An exercise in proving parallel programs correct. Comm. ACM 20, 12 (Dec. 1977), 921-930.

13. Henderson, P. Functional Programming, Application and Implementation. Prentice-Hall International, London, 1980.

14. Knuth, D.E. The Art of Computer Programming, Vol. III: Sorting and Searching. Addison-Wesley, Reading, MA, 1973, p. 522.

15. Lamport, L. Proving the correctness of multiprocess programs. IEEE Trans. Software Eng. SE-3, 1 (March 1977), 125-143.

16. Landin, P.J. The next 700 programming languages. Comm. ACM 9, 3 (March 1966), 157-162.

17. McCarthy, J. A basis for mathematical theory of computation. In Computer Programming and Formal Systems, P. Braffort and D. Hirschberg, Eds., North-Holland, Amsterdam, 1963, pp. 33-70.

18. Owicki, S., and Gries, D. Verifying properties of parallel programs: an axiomatic approach. Comm. ACM 19, 5 (May 1976) 279-285.

19. Shaw, A.C. The Logical Design of Operating Systems. Prentice-Hall, Englewood Cliffs, NJ, 1974, pp. 80-82.

20. Tanenbaum, A.S. Structured Computer Organization. Prentice-Hall, Englewood Cliffs, NJ, 1976.

21. Tesler, G., and Enea, H.J. A language design for concurrent processes. In Proc. Spring Joint Computer Conference, Thompson, Washington, 1968, pp. 403-408.

22. Thurber, J. Interprocess communication: hardware interconnection technology. Advanced Course on Distributed Systems Architecture and Implementation, 1980 Institute für Mathematik und Informatik Technische Universitat, München, March, 1980.

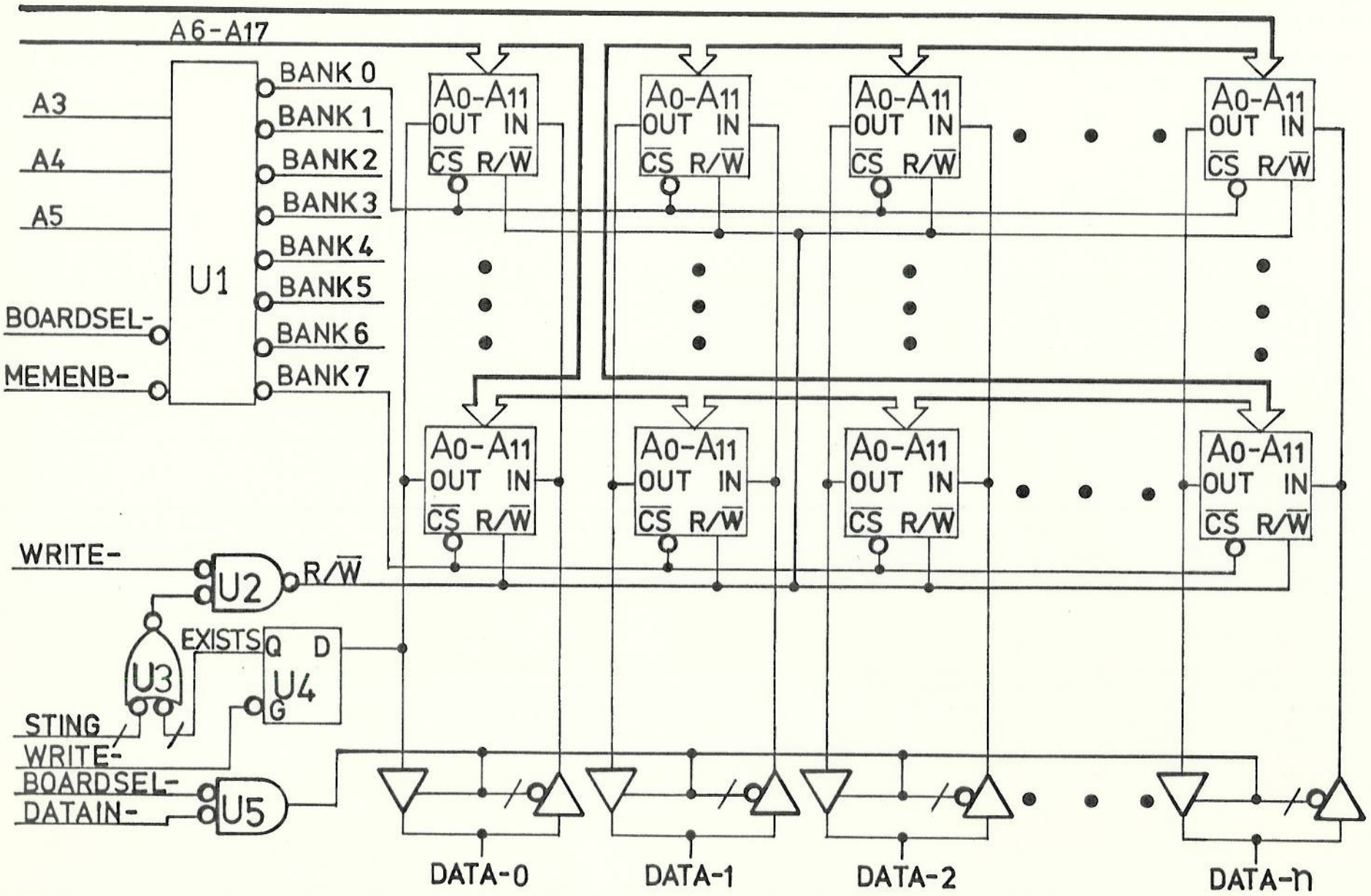
A6-A17
A3
A4
A5
BOARDSEL-
MEMENB-
U1
BANK 0
BANK 1
BANK 2
BANK 3
BANK 4
BANK 5
BANK 6
BANK 7
A0-A11
OUT IN
CS R/W
WRITE-
U2
R/W
EXISTS
Q D
U4
G
U3
STING
WRITE-
BOARDSEL-
DATAIN-
U5
DATA-0
DATA-1
DATA-2
DATA-n

Figure 1.