MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

Computation Structures Group Memo No. 56

Asynchronous Arbiters

William W. Plummer

February 1971

This research was done at Project MAC, MIT, and was supported in part by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Nonr N00014-70-A-0362-0001.

ASYNCHRONOUS ARBITERS

William W. Plummer

Abstract -- When two or more processors attempt to simultaneously use a functional unit (memory, multiplier, etc.), an arbiter module must be employed to insure that processor requests are honored in sequence. The design of asynchronous arbiters is complicated because multiple input changes are allowed, and because inputs may change even if the circuit is not in a stable state. A practical arbiter and its implementation are presented. Implementation of various priority rules (linear, ring, mixed) is discussed, and building large arbiters with trees of two-user arbiters is considered.

Index Terms -- asynchronous arbiter, asynchronous logic design, hardware resource allocation, multiprocessor computer system, functional unit allocation, sequential machines with multiple input changes, priority network, conflict resolution, modular control logic, macromodules.

I. Introduction

In a parallel system two or more processors may (almost or exactly) simultaneously request the use of a particular resource. Thus, it is necessary to resolve the conflict and to allocate the resource first to one processor and then to the next, in sequence, until all requests have been serviced. During the time that one request is being handled, more requests may appear.

Figure 1 contains two instances of such conflict. The first instance is at the multiplier which is shared by two processors. If either processor alone executes a multiply instruction, no problem exists. However, if both simultaneously request the use of the multiplier, the ARBITER logic allows one processor to access the multiplier while the request from the other processor is left pending for future action.

At the bottom of Figure 1 both processors and the data channel [illegible] access to the memory system. A typical sequence of activities will proceed as follows: initially both processors are active and the [illegible]. The memory will be allocated to one of the processors, [illegible] finishes an instruction that causes the data channel to request a word from memory. The arbiter will, because of its priority structure, service the channel next and finally go on to the other processor's request. Figure 2 displays this activity.

In Figure 1 the multiplier and the memory were accessed through arbiters which resolved any conflicts. This paper is concerned with the detailed implementation of arbiter circuits, primarily asynchronous ones. It is assumed that each port through which the arbitrated resource may be accessed consists of a request wire and an acknowledge wire as shown in Figure 3. An "up" transition of a Request signal initiates activity which terminates by sending an up transition on the corresponding Acknowledge wire. The module driving the port then resets it by sending a "down" transition on the Request wire. When the port is idle, a matching down transition will be received on the Acknowledge wire.

II. Requirements of the Arbiter

A proper arbiter implementation will obey the following rules:

1) With some finite delay, exactly one output request R_0 must be generated for each input request R_i. That is, the arbiter may not create output requests corresponding to phantom input requests, nor may it destroy input requests without servicing them.

2) The arbiter should not begin servicing a port until it has completely reset the most recently serviced one. This assures that two cycles will not be done for one request and disallows any "overlap" of the next request with the previous reset.

3) The ports do not intercommunicate. Activity on one port can only delay service to some other port, and cannot prevent a request from occurring on the other port. Thus, the arbiter must be able to accept a request from any idle port at any time.

III. Overall Structure

The arbiter to be described has the overall structure shown in Figure 4. Each input port is connected to a block which emits an ACTive level that indicates that this port is either requesting the server or that it is being serviced. This block also generates a modified request signal, R".

The control section is activated by the OR of all ACT_j levels and generates the following signals:

name	true when:
AWAIT	The arbiter is free to consider another request.
DECIDE	The arbiter is deciding which of several requesting ports to service.
R_0	Server is being requested on behalf of some port.
ACK	Resetting the port just serviced.

In addition, the OR of the R''_j signals is used in forming R_0.

The priority network implements a rule by which the arbiter decides which port to service if several are requesting. It effects this decision by using the PRIority signals to turn off ACTive signals on all ports which are not going to be serviced. This action occurs when DECIDE is true. The priority network may remember the history of service to the various ports in order to do its job. For instance, a 3-port arbiter which tries to maintain service ratios of port1:port2:port3::3:2:1 (in the fully loaded case) must remember which port was serviced on each of the past six cycles. From this, the "current service ratios" may be known and priority resolved in a way that makes the current service ratios approach the designed-for 3:2:1.

IV. Detailed Logic Design

A. The Procedure

Design of arbiters is somewhat harder than most logic circuits because traditional design approaches are vastly too cumbersome. The usual design assumptions are that inputs are allowed to change only if the circuit is in a stable state and that only one input at a time will change. Arbiters violate both of these. The technique employed here relies heavily on experience and intuition, but breaks down into the following steps: formalizing the problem and establishing signalling conventions (requests, acknowledge signals and the rules of how they are to act), deciding what the required actions to be performed are (noting input requests, deciding priority, activating the server, and handling the server completion), and designing "small" circuits which perform these actions in a consistent way. No claim is made for the minimality of the circuits presented here. Indeed, several of them have extra gates which have been included to improve the clarity. All circuits are realized with NAND gates.

B. Port Logic

Associated with each input port of the arbiter is a section called the "port logic". Each of these sections has two parts, the ACTive flip-flop and associated gating, and a "buffer" circuit. This is shown in Figure 5

The properties of the buffer section are shown in the graph in Figure 6. In graphs of this type, a solid arrow from x to y means that the circuit being described must emit the event y immediately after receiving the event x. A dashed arrow from x to y means that if the circuit emits an x event, it will receive a y event in response some arbitrary time later. There are two kinds of events: signals becoming true, denoted by an instance of the signal name without an overbar, and signals becoming false, in which case the signal name will bear an overbar.

Referring to Figure 6, an R_j becoming true makes R'_j true. Sometime later A'_j becomes true, ending the information transaction. This causes R'_j to become false (initiates a reset transaction to the ACTive logic) and A_j to become true (request a reset transaction from the logic driving port j). When the ACTive logic and the server have been reset (A'_j is false) and a reset transaction has been requested on the port (R_j is false), the A_j is made false, terminating the cycle.

The logic surrounding the $ACTive_j$ flip-flop is described in Figure 7. The ACT_j flip-flop is set if there is a request on port j and the arbiter is AWAITing a request. ACT_j may be cleared by PRI_j during the priority decision time if a port of higher current priority is to be serviced. Sometime later a cycle will occur for $port_j$ and the server's acknowledge will be directed to this port (A'_j, A_j). Eventually, the reset request will arrive ($\overline{R_j}$, $\overline{R'_j}$) and this will initiate a reset request to the server ($\overline{R_0}$). When the server has been reset ($\overline{A_0}$), the reset acknowledge ($\overline{A'_j}$, $\overline{A_j}$) is issued and ACT_j cleared. This latter action will cause the arbiter to AWAIT another request.

Each port logic section generates a signal R''_j which is used by the control part in forming the output request R_0. R''_j is the AND of R'_j and ACT_j and indicates that the server is cycling for $port_j$ and that the reset request has not occurred yet. When R_j becomes false, so does R'_j. Then R''_j becomes false causing R_0 to do the same, but leaving the ACT_j flip-flop on so that A_0 can be directed back to the correct port.

C. Control Section

A diagram of the control section is shown in the lower right corner of Figure 11. It contains the AWAIT flip-flop, DECIDE one-shot and gates to produce the output request R_0. The control is first activated by an ACT_j signal becoming true. It is important to note that, although several ACT_j signals may become true at the same time, the fact that the control has been activated guaranttes that at least one ACT_j is true. The function of the control is to do the following sequence of operations:

1) Make AWAIT false so that no more ACT_j signals may be set.

2) DECIDE, using the priority network, which one of the ACT_j signals should remain on for service. Clear those which will not be serviced by this cycle.

3) After (2) has been accomplished, exactly one ACT_j signal will be true, and this represents the port to which the acknowledge (A_0) will be directed. So, step (3) consists of generating an output request R_0.

4) When A_0 occurs, it is directed back to the port being serviced, when then lowers its R_j, which initiates the reset cycle as previously described.

5) When the $port_j$ is idle, the port logic will clear its ACT_j flip-flop. The control section detects the condition where all ACTive flip-flops are off, it sets AWAIT so that another request may be processed.

It is important to note that step (2) requres a non-zero amount of time, the length of which is determined by the one-shot. This must be long enough to let the priority network settle down after the last change in the ACT_j flip-flops. In a synchronous arbiter this will be the time between two clock pulses, the first of which strobes the requests, and the second does the priority selection and output request generation. The asynchronous arbiter defines these two time instants by the on and off transitions of the DECIDE one-shot.

D. Priority Networks

1. Introduction

During DECIDE, the arbiter must turn off all but one ACTive latch. The priority network implements some rule by which this one, highest priority port is selected. Of course, any priority scheme must allow all requests to be serviced eventually, and the usual assumption is that the lowest priority

port(s) is requesting continually -- the "fully loaded case."

As mentioned previously, the priority network may keep a service history so that it can implement complicated priority rules. Usually, however, a simple "ring" or "rotating" rule is desired and this requires knowledge of only the single, most recently serviced port. Sometimes the even simpler "linear" priority rule is selected, which requires no memory at all.

2. Linear Selection

Figure 8 shows the logic for the linear selection priority rule. If any ACT_i signal becomes true, all those below it are forced off. Thus, the PRI_i signal is just the OR of all ACT_j signals where j is less than i. Note that the PRI_i signals are not gated by DECIDE. This is permitted for linear select networks because they have no memory which needs to be updated. A priority network which has states may be thought of as implementing two different selection rules -- one before the memory is updated, and the other after it is updated. In such circuits the PRI_i signals must be gated with DECIDE so that priority is resolved using the current contents of the memory, and then the memory is changed.

3. Ring Selection

Ring selection is a generalization of linear selection. At any instant the network is actually a linear selection network, the end of which is specified by the contents of a register (CT) which holds the number of the last port serviced. This is the lowest priority port for the next cycle, and the port next in line around the ring will have the highest current priority.

The priority is resolved during DECIDE. The trailing edge of DECIDE strobes the encoded port number (about to be serviced) into the memory register. Thus, at the beginning of the next cycle this will contain the number of the port serviced on the last cycle, as required. Figure 9 shows the general form of a ring priority network. In this case the CT register is binary coded. It is possible to use a unary coded scheme (N flip-flops and no encoder or decoder) but care must be taken to insure that the arbiter is

... so that exactly one of the N bits is on.

Ring priority is useful because it gives all ports an equal [illegible] ... service. In effect it is a round-robin scheduler since the ports ... serviced in order around the ring. No port can prevent others from ... by continually requesting.

4. A Mixed Priority Scheme

Sometimes it is desirable to have arbiters with mixed priority rules. Figure 10 shows a network that gives port 1 precedence over all others, port 2 precedence over ports 3 and 4, and ports 3 and 4 equal and alternating (ring) priority.

An application for an arbiter with such a priority network might arise at the memory of a computer which has two instruction processors (ports 3 and 4), a data channel (port 2), and a drum (port 1). It is clear that the instruction processors can both be given the lowest priority because instruction execution can be delayed indefinitely with no ill effects. Alternating priority is given to the processor ports so that neither can inhibit the progress of the other by continuously requesting the memory. A drum on the other hand, is a high rate device which cannot be stopped if the memory is not available. Therefore, it is connected to the highest priority port. All other devices (tapes, displays, etc.) can be multiplexed into port 2.

V. Arbiter Trees

Figure 11 is the complete diagram of a two input arbiter with ring priority. Three of these may be interconnected to form a four port arbiter (Figure 12a). In the fully loaded case the ports might be serviced in the order 1, 3, 2, 4, 1, 3, 2, 4, ... The interesting point is that even though an arbiter requires a long delay (DECIDE), and this construction of a four port arbiter from three 2-port arbiters has two levels, it operates at the same rate as a single four port arbiter with the same priority rule, assuming that DECIDE is shorter than the time the server stays busy. To understand

this, assume that while port 1 is being serviced, the arbiter with inputs from ports 3 and 4 will be DECIDE-ing which of those will be next. Thus, when port 1 is done, the main arbiter will already have a request on behalf of port 3 or port 4. Consequently, there is always an overlap -- one of the secondary arbiters making a decision, while the other is in use. The primary arbiter is guaranteed to always have a request waiting to be serviced. A wide class of priority networks can be simulated exactly or approximated by trees of two input arbiters. Figure 12b has almost the same priority behavior as the network in Figure 10. Thus, the two input arbiter (diagram in Figure 11) is a basic building block for more complicated arbiters.

VI. Acknowledgement

The author thanks Prof. J. B. Dennis, Prof. Suhas Patil, Mr. John A. McKenzie, and Miss Anne Rubin, who all contributed to this paper.

VII. References

[1] D. A. Huffman, "The synthesis of sequential switching circuits," J. Franklin Institute, vol. 257, pp. 161-190, 275-303, March and April 1954.

[2] D. A. Huffman, "Design and use of hazard-free switching networks," J. ACM, vol. 4, pp. 47-72, January 1957.

[3] S. H. Unger, "Hazards and delays in asynchronous sequential switching circuits," IRE Trans. Circuit Theory, vol. CT-6, pp. 12-25, March 1959.

[4] G. A. Maley and J. Earle, The Logical Design of Transistor Digital Computers, Englewood Cliffs, N. J.: Prentice-Hall, 1963.

[5] R. McNaughton, "Badly timed elements and well timed nets," Moore School of Engineering, Rept. 65-02, June 10, 1964.

[6] E. J. McCluskey, Introduction to the Theory of Switching Circuits, New York: McGraw-Hill, 1965.

[7] R. E. Miller, Switching Theory, vol. 2, New York: Wiley, 1965.

[8] D. B. Armstrong, A. D. Friedman, P. R. Menon, "Design of asynchronous circuits assuming unbounded gate delays," IEEE Trans. Computers, vol. C-18, pp. 1110-1120, December 1969.

[9] D. H. Bredt and E. J. McCluskey, "A model for parallel computer systems," Stanford Digital Systems Laboratory Rept. 5, April, 1970.

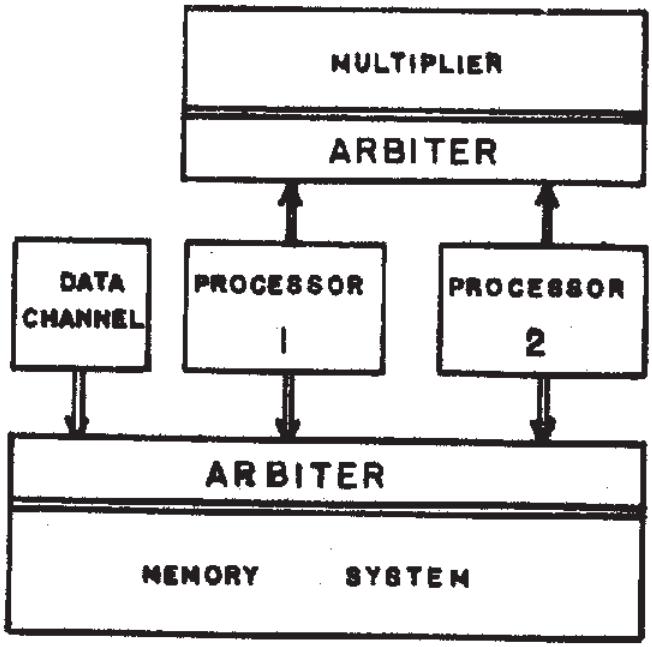

FIGURE 1. CONFLICT SITUATIONS

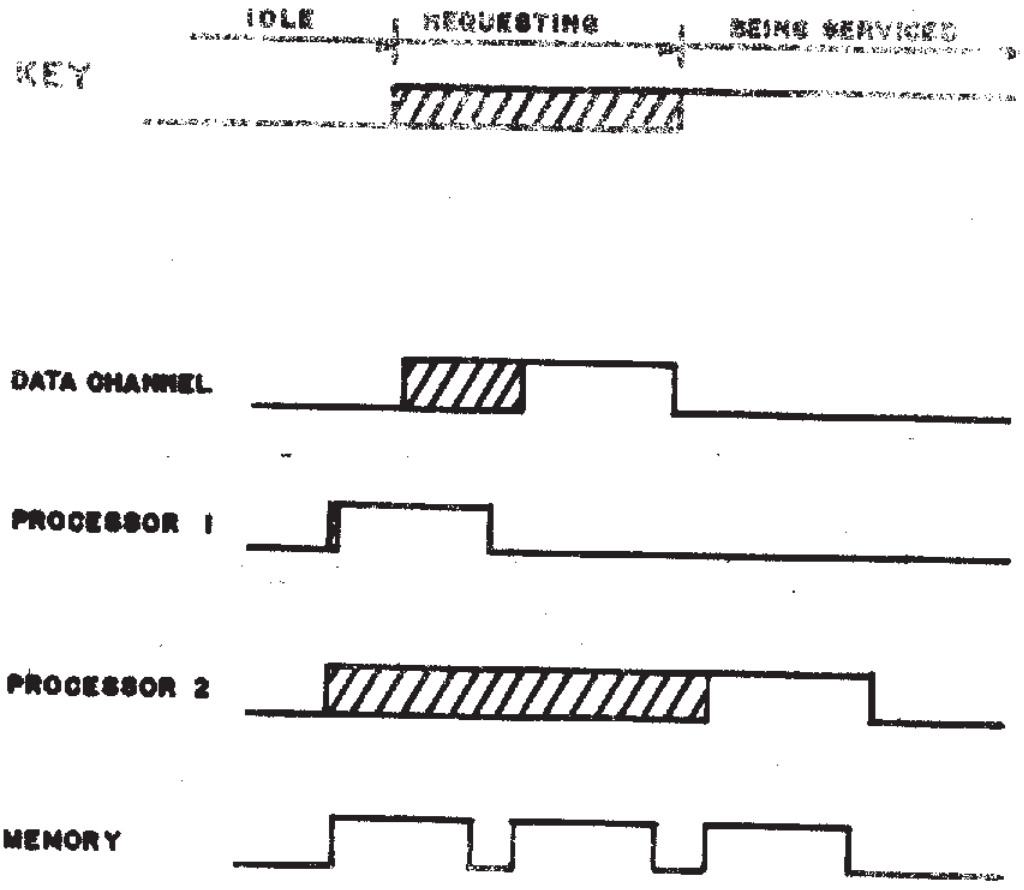

FIGURE 2. TYPICAL ACTIVITY IN FIGURE 1.

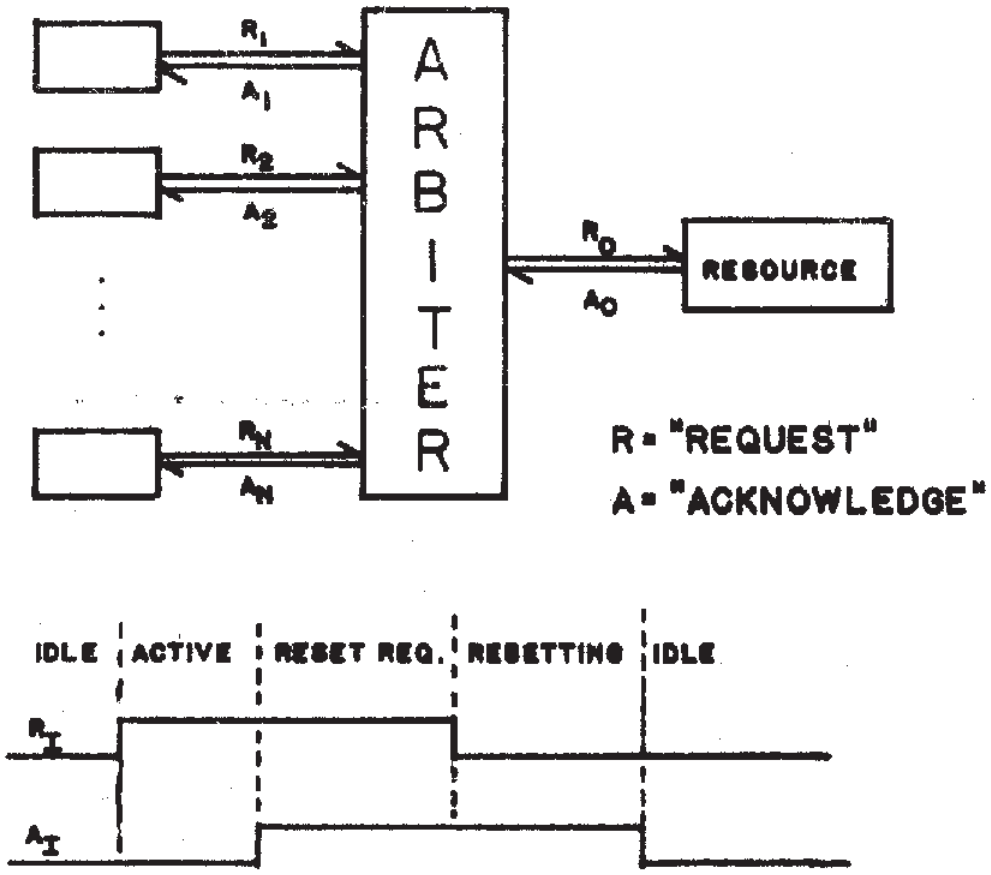

FIGURE 3. SIGNAL CONVENTIONS

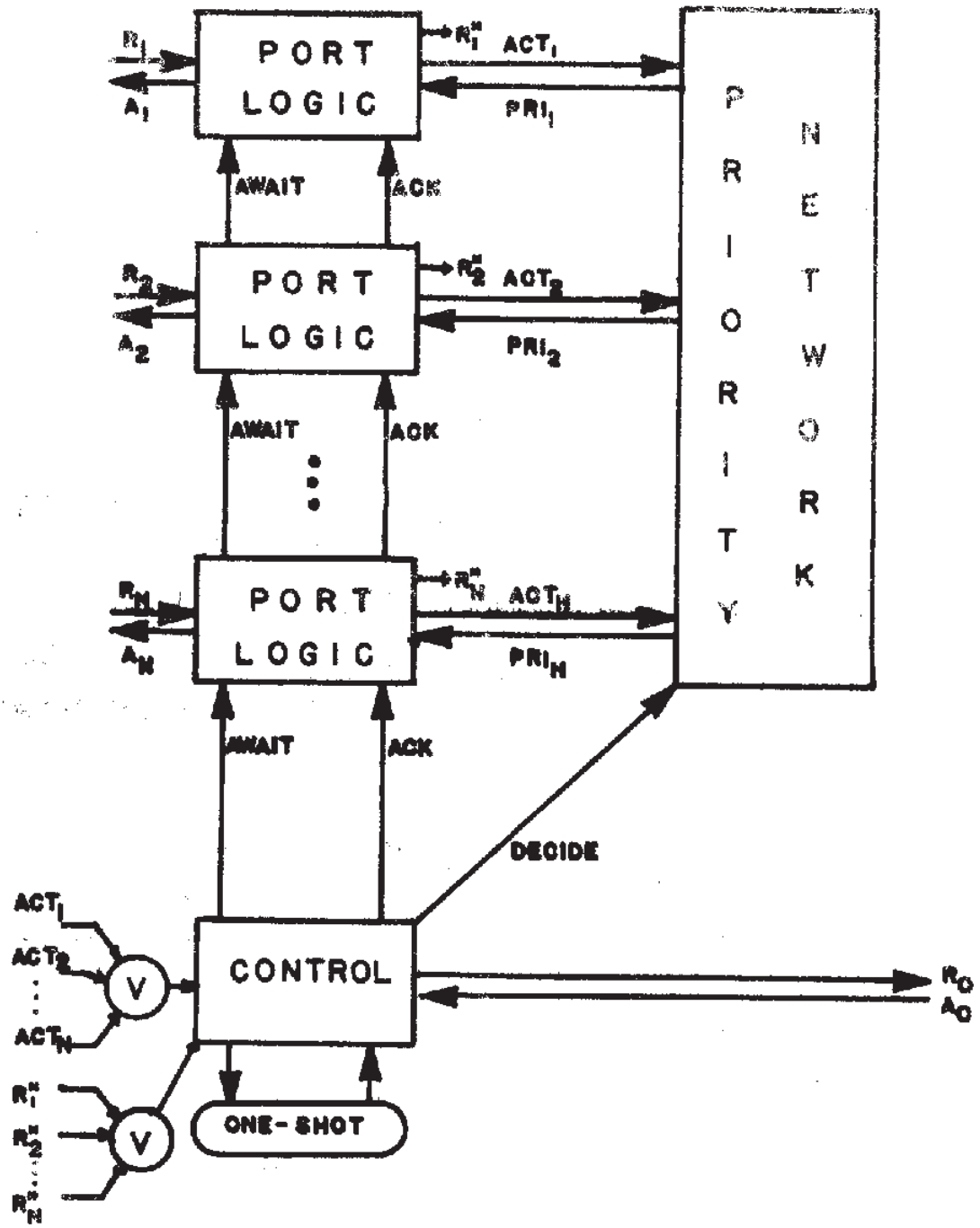

FIGURE 4. ARBITER BLOCK DIAGRAM

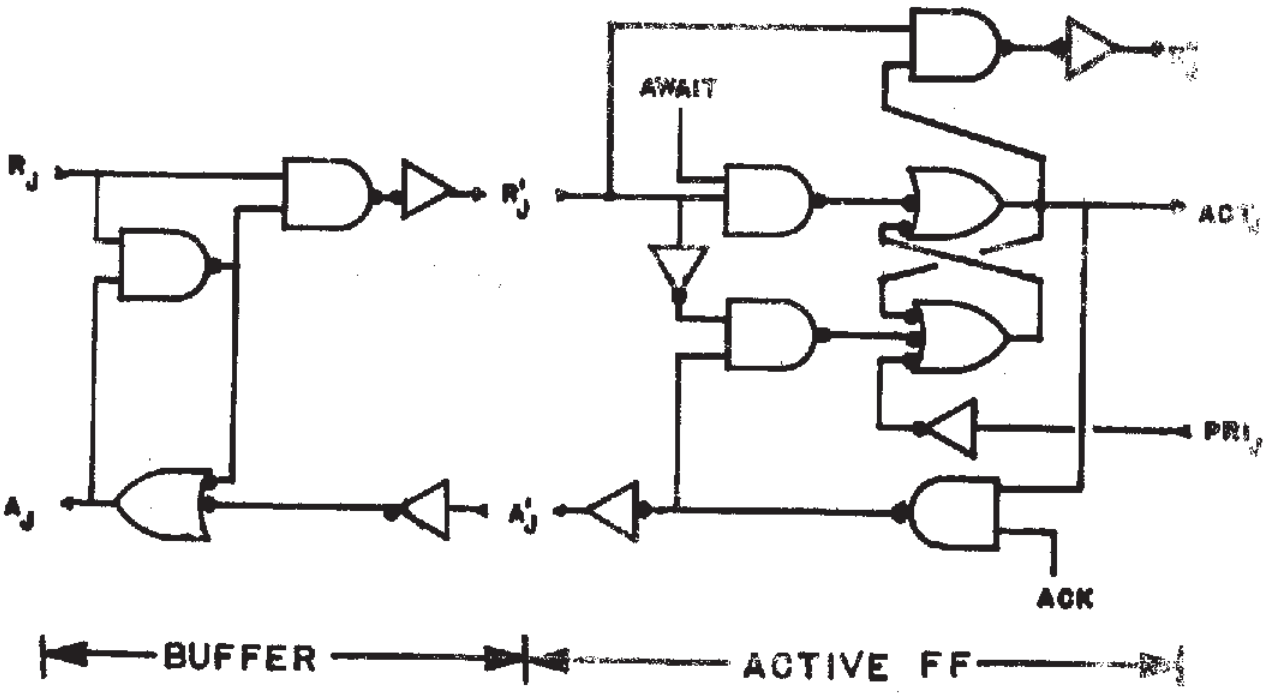

FIGURE 5. PORT LOGIC

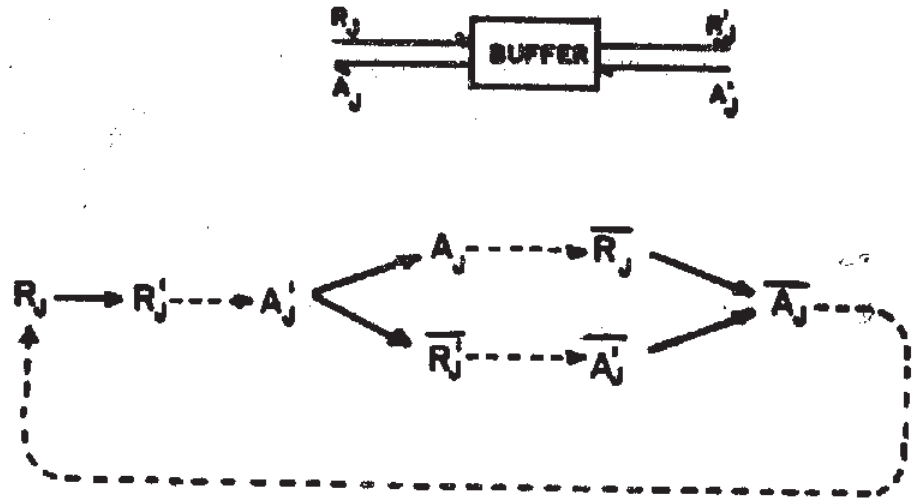

FIGURE 6. BUFFER ACTION

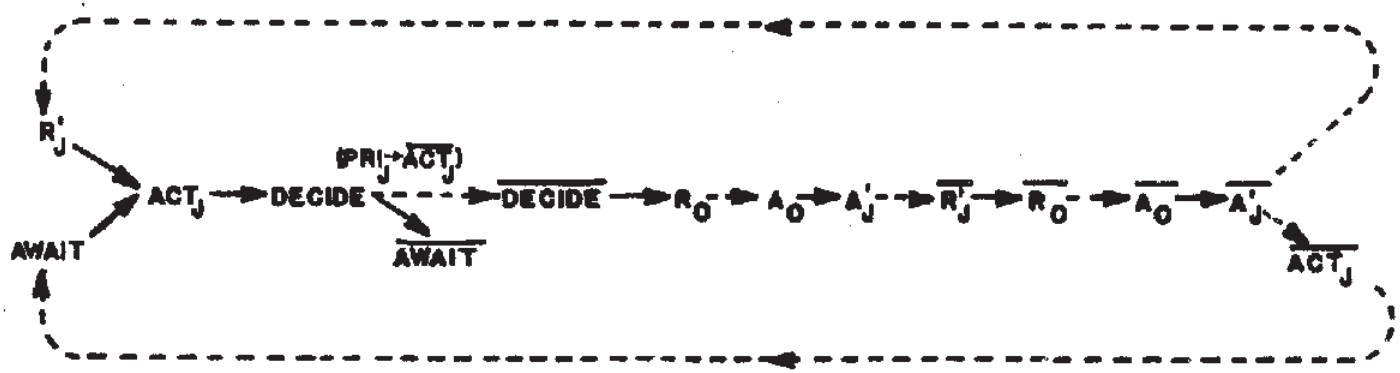

FIGURE 7. PORT LOGIC ACTION

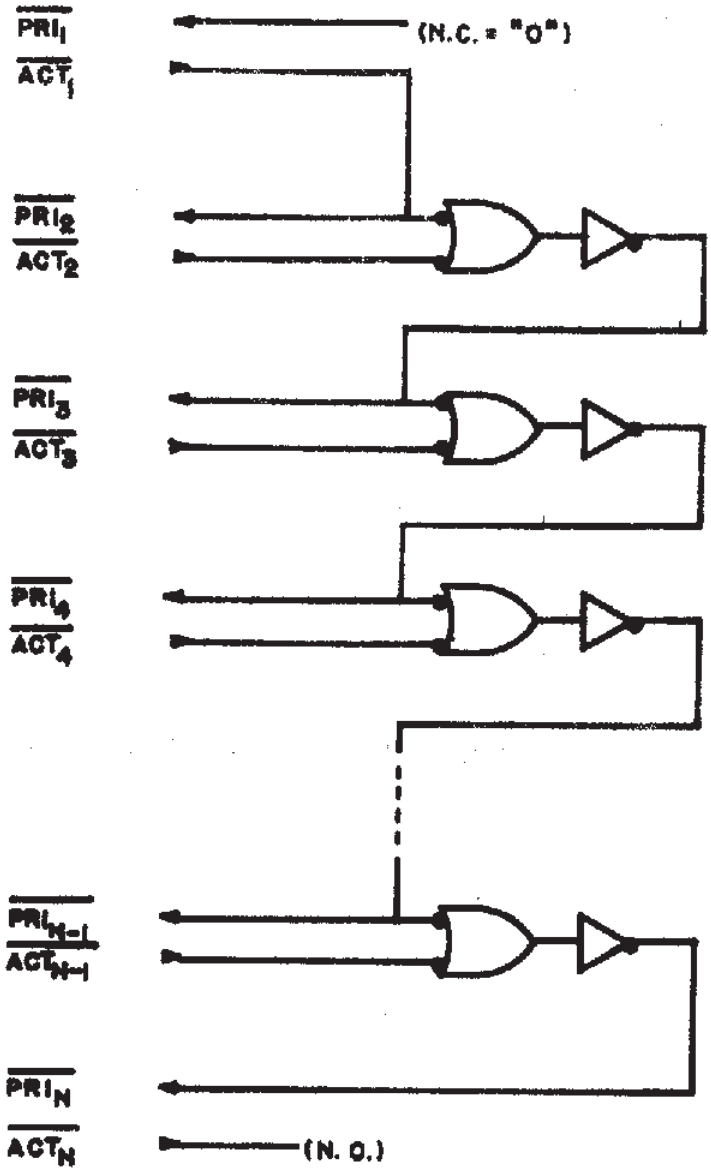

FIGURE 8. LINEAR PRIORITY SELECTION NETWORK

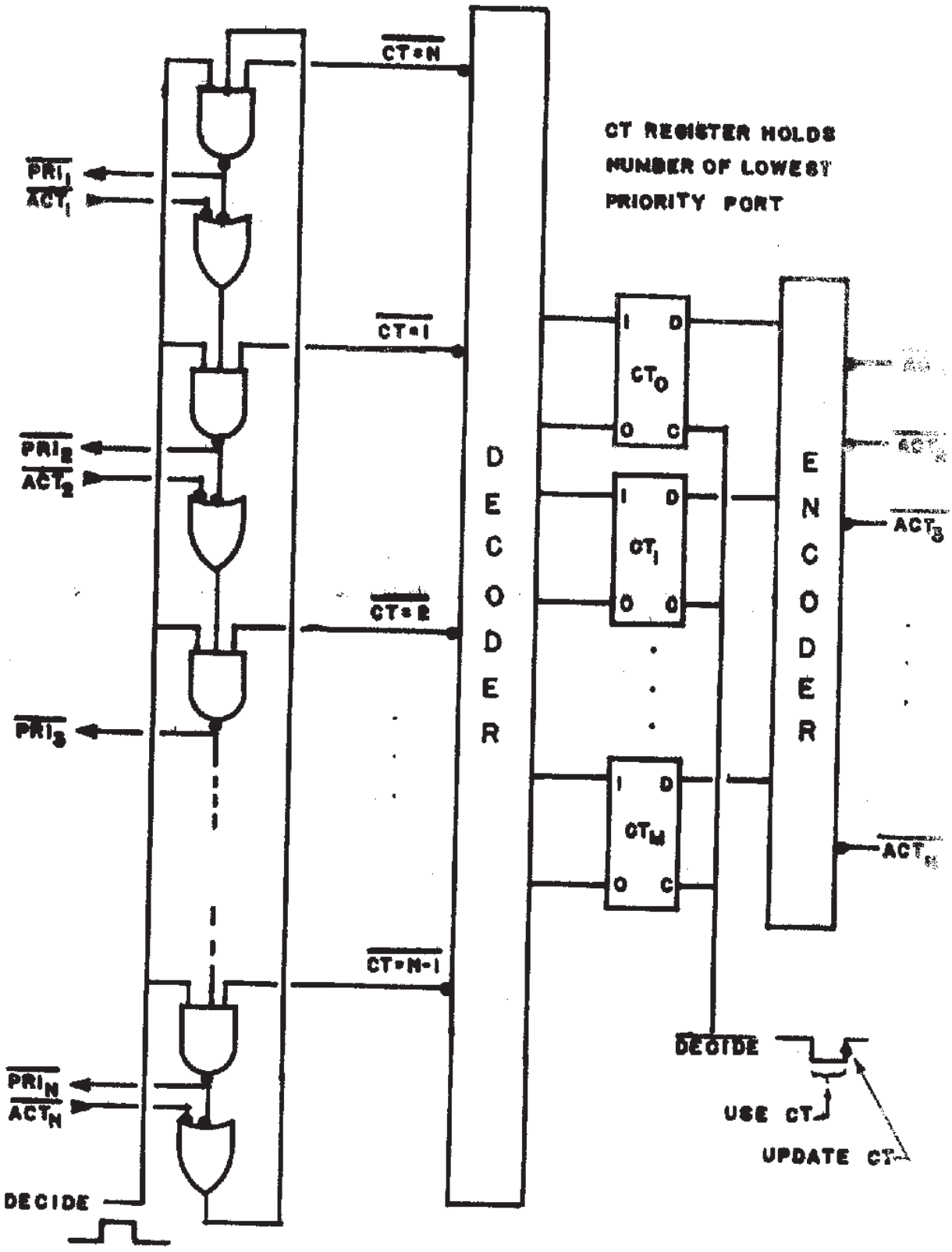

FIGURE 9. GENERAL RING PRIORITY NETWORK

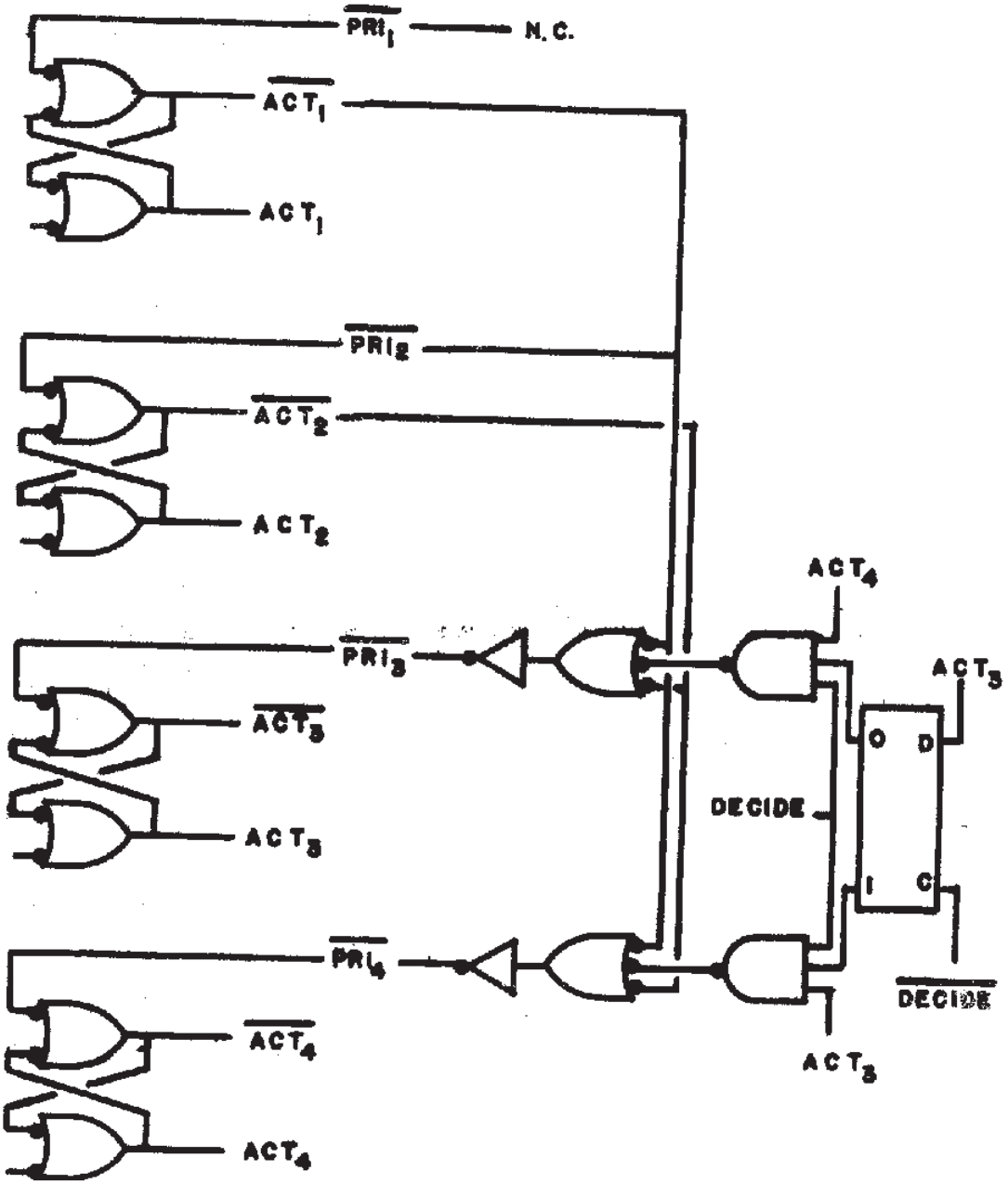

FIGURE 10. A MIXED PRIORITY SCHEME

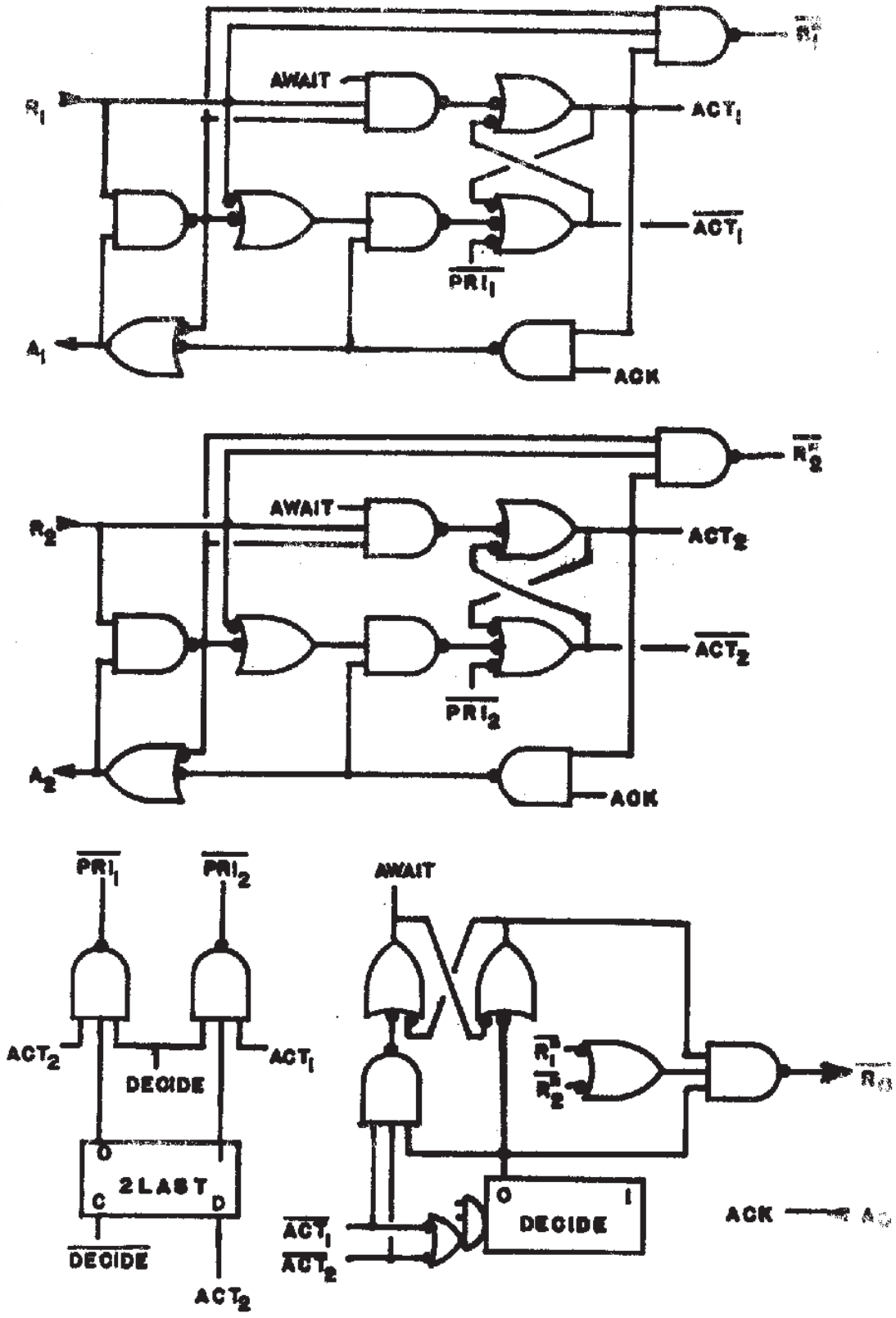

FIGURE 11. COMPLETE TWO INPUT ARBITER

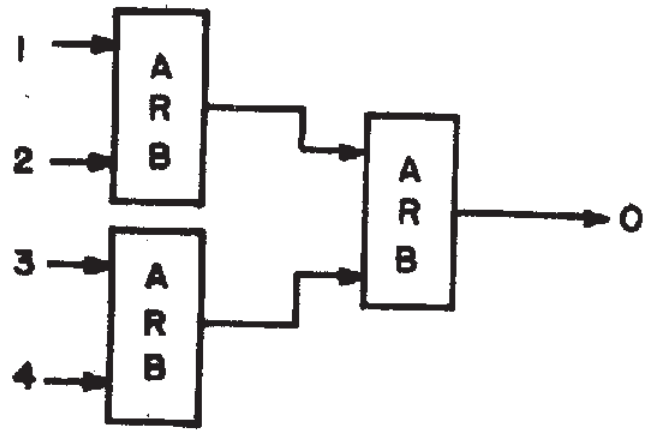

FIGURE 12 A. ARBITER TREE WITH RING PRIORITY

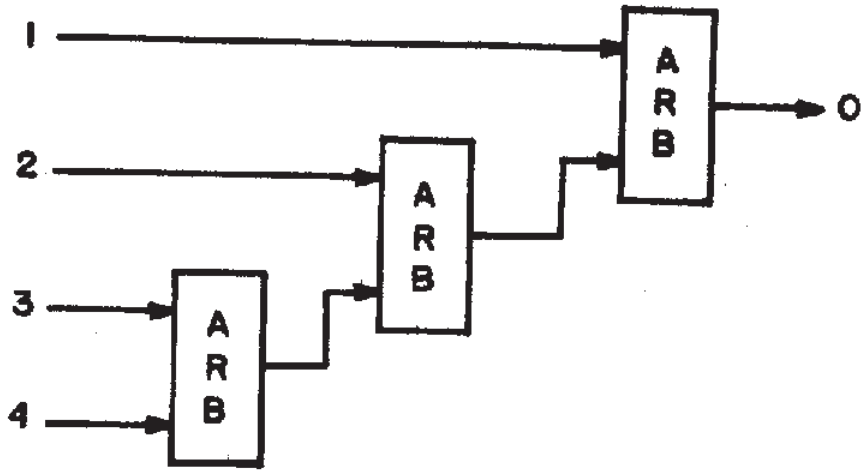

FIGURE 12 B. ARBITER TREE WITH ALMOST THE SAME PRIORITY AS THAT IMPLEMENTED BY THE NETWORK IN FIGURE 10.