IDEAL ARBITERS: ANALYSIS AND DESIGN

Kostas N. Oikonomou

Kostas N. Oikonomou *is a member of technical staff in the Data Communications Planning Department of AT&T Bell Laboratories in Holmdel, New Jersey. He is working on modeling and performance analysis of forward-looking wideband network technology. Mr. Oikonomou, who joined Bell Labs in 1981, has a Ph.D. in electrical engineering from the University of Minnesota.*

An *ideal* arbiter for concurrent processes that share a resource ensures the mutual exclusion of the users, is fair, and grants requests in the order that they were made. Non-ideal arbitration schemes, both of algorithmic and circuit form, with the first two properties are well-known. Here we study ideal arbiters in a top-down manner, at three levels of abstraction: behavioral properties expressed in temporal logic, discrete-time automata, and Boolean circuits and functions. We show that an n-process arbiter can be realized with complexity $O(n^2 \log n)$ and delay $O(\log n)$. The goodness of these results is examined from a viewpoint of Boolean complexity theory. We also study a two-level modular realization of the arbiter, equivalent to the above monolithic one. If the equivalence is not required to hold under worst-case user behavior, the modular arbiter can be implemented with much smaller complexity than the monolithic one and with comparable delay.

Introduction

An arbiter is a device that allows several concurrent processes to share a single resource correctly and efficiently. For example, an arbiter can control access to the bus that connects the processors of a multiprocessor system to a shared memory. An *ideal* arbiter has the following logical properties:

1. It eventually grants every request, and does not issue grants without prior requests (*fairness* and converse).
2. It lets only one process access the resource at any time (*mutual exclusion*).
3. It grants requests in the order that they were made (*first-come, first-served*, FCFS).

A performance-related property of the arbiter is that it grant requests as quickly as possible.

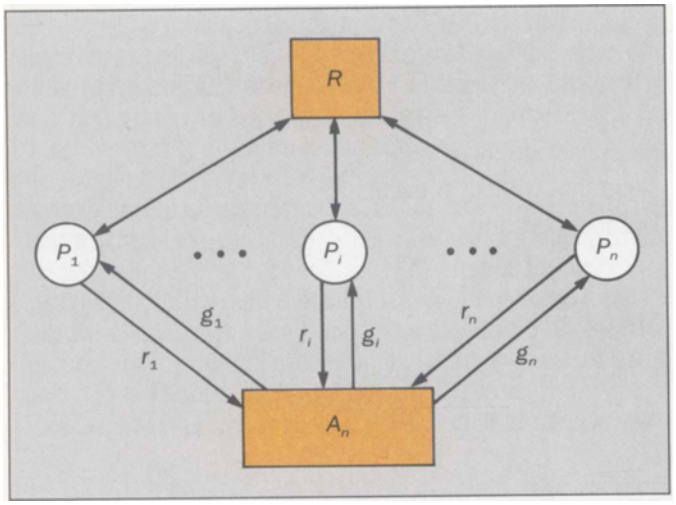

Figure 1. Processes $P_1, \ldots, P_n$ share a resource R that is controlled by an arbiter A_n.

The arbitration problem is well-known in the algorithm (software) domain under the name "mutual exclusion." Deterministic and probabilistic algorithms have been proposed for this problem, for both shared-memory (centralized)[1] and message-based (distributed)[2] systems. Also, several circuit realizations of arbiters—both synchronous[3,4] and asynchronous[5] designs—have been studied, and all have properties 1 and 2. However, to the author's knowledge, an ideal arbiter that also has the FCFS property (3) has not been investigated in any depth.

In this paper, we approach the analysis and design of ideal arbiters in a *top down* manner, using three levels of abstraction: behavioral specifications (expressed in temporal logic), discrete-time automata, and Boolean functions and circuits.

First, the behavior of the arbiter desired by the users is specified precisely using the formalism of linear-time temporal logic. This specification says nothing about how to realize the arbiter. For the temporal specifications to be met, user processes must obey a simple protocol that consists of a single requirement: requests cannot be repeated until after they are granted.

Next, we model the n-process arbiter by a finite-state, discrete-time automaton A_n, whose input-output behavior is consistent with the high-level temporal specifications. For this discrete-time (synchronous) model to be applicable, the users' requests must be *synchronized* to the arbiter's clock. This is a subtle problem[6] that is not dealt with here. We show that A_n has about $e\,n!$ states, no two of which are equivalent. About $(e - 1)\,n!$ of these states are reachable from the automaton's initial "empty" state, and all are strongly connected. The large number of densely interconnected states is some evidence that implementing A_n as a single circuit may be difficult for large n.

As a first step in investigating how to realize A_n from smaller modules, we show that certain straightforward modularizations are impossible simply because the number of states of A_n grows too fast with n. To modularize an n-input arbiter, for $n = mM$, we define a new automaton B_M that *coordinates* M automata A_m so that their behavior is identical to that of A_n.

Finally, we study the *Boolean functions* needed to implement the arbiter automaton from the viewpoint of reducing the complexity (number of gates needed), while maintaining high speed. We do not address the problem of implementing these functions in VLSI (very-large-scale integration). It is shown that A_n can be realized with complexity $O(n^2 \log n)$ and delay $O(\log n)$, and that less than logarithmic delay cannot be practically achieved. It is also shown that, under some assumptions on the user behavior, the modular arbiter can be implemented with complexity of only $O(n^{4/3} \log n)$ and delay again $O(\log n)$.

Specification of the Arbiter System

Figure 1 shows n user processes $P_1, \ldots, P_n$ that share a resource R through an arbiter A_n. To communicate, the users and the arbiter exchange *request* and *grant* sig-

Panel 1: Temporal Specifications for the Arbiter

We assume that the reader is somewhat familiar with temporal logic and use the following notation:

- $\vdash \phi$ means that, if the system is properly initialized, then *all* its possible behaviors satisfy formula (have property) ϕ.
- $\wedge$, $\vee$, $\supset$, and $^-$ are the Boolean connectives *and, or, implies,* and *not,* and $\equiv$ denotes equivalence ($\supset$ and $\subset$).
- $\Box$, $\Diamond$, $\bigcirc$, and P are the temporal operators *always, eventually, next,* and *precedes* as defined in reference 7. For example, $r\mathsf{P}g$ means that, if g is true at some instant $t_g \geq 0$ of discrete time, then r is true at some instant $0 \leq t_r < t_g$. Finally, $\uparrow x$ abbreviates $\overline{x} \wedge \bigcirc x$, and $\downarrow x$ abbreviates $x \wedge \bigcirc \overline{x}$.

Initial conditions

0. $\vdash \bigwedge_i \overline{g}_i$

Local specifications

1. $\vdash \Box (r_i \supset \Diamond g_i)$ — Responsiveness/fairness
2. $\vdash r_i \mathsf{P} g_i \wedge \Box (\downarrow g_i \supset \bigcirc (r_i \mathsf{P} g_i))$ — No grants without requests
3. $\vdash \Box (g_i \supset \bigcirc \overline{g}_i)$ — Duration of grant

Global specifications

4. $\vdash \Box (g_i \supset \bigwedge_{j \neq i} \overline{g}_j)$ — Mutual exclusion
5. $\vdash \Box (\overline{g}_j \supset (r_i \mathsf{P} r_j \supset g_i \mathsf{P} g_j))$ — First-come, first-served
6. $\vdash \Box (\bigvee_i p_i \supset \bigcirc \bigvee_i g_i)$ — Speed of response

Protocol specification

7. $\vdash \Box (r_i \supset \bigcirc (g_i \mathsf{P} r_i))$ — Requests can't be repeated until after granted

Auxiliary specifications

A0. $\vdash \bigwedge_i \overline{p}_i$ — p_i: r_i is "pending"

A1. $\vdash \Box (\uparrow r_i \equiv \uparrow p_i \wedge \uparrow g_i \equiv \downarrow p_i)$

nals r and g. When P_i wants the resource, it signals its request on r_i. When, some time later, A_n signals on g_i (grants the request), then P_i can use R for one time unit without interference.

For simplicity, we assume that R is always available whenever the arbiter grants it to some user process. That is why there is no communication between R and A_n in Figure 1. Whether this assumption is realistic depends on what the shared resource R is.

Temporal Specifications. At the highest level of abstraction, the arbiter in Figure 1 is specified by the behavior that the *user processes* see. This behavior is a *set of sequences* of request (input) and grant (output) signals. To define precisely the input-output sequences that are acceptable to the users, we employ the formalism of linear-time *temporal logic*;[7] see Panel 1.

The *initial conditions*, (0) in the panel, specify that, at $t = 0$, there are no grants in the system. In interpreting the rest of the formulas, two things have to be kept in mind:

- (7) and (3) mean that the request and grant signals are *pulses*, one time unit wide.
- Except for (0) and (4), the temporal formulas express a user's informal requirements correctly *only if* the protocol (7) is obeyed. (See reference 8, section 4.2 for details.)

The *local* specifications in Panel 1 relate a request signal r_i to the corresponding grant signal g_i and express how a user views the arbiter. Formula (1) says that every request will eventually be granted, while (2) states that, conversely, there will be no grants without prior requests. Formula (3) simply says that a grant lasts one time unit.

The *global* specifications in Panel 1 state relationships among all the r_i and g_i, thus expressing the viewpoint of a (hypothetical) global observer. Formula (4) says that only one grant can be true at any instant (mutual exclusion). Formula (5) requires that requests be granted in the order that they arrive: If r_i precedes r_j, then g_i should occur before g_j. This precedence property should be tested only when g_j is false. Formula (6) says that as long as there are pending requests, one should be satisfied at every (next) instant. To define what it means for r_j to be pending, we use the auxiliary Boolean proposition p_i (a signal not present in the real system), specified by formulas (A0) and (A1) in Panel 1. (A1) says that p_i becomes true when r_i does, remains true until g_i becomes true, and then turns false.

The *protocol* specifications are the *constraints* that user processes must meet if the arbiter's behavior is to have properties (1) through (6). Formula (7) expresses our single protocol requirement: Once a request is made, it cannot be repeated until after it is granted. This requirement has two effects. First, it limits the number of requests that the arbiter has to remember. Second, it makes possible an unambiguous one-to-one correspondence between r_i and g_i.

Our very simple protocol is based on two assumptions:

- The shared resource R is always available.
- Every process uses R for a single time unit.

These assumptions are too unrealistic for some applications, but are sound if R is, for example, a high-speed bus (this application is discussed in reference 3.)

Arbiter protocols of greater logical complexity have been proposed. For example, the *resource granter* protocol that Manna and Pnueli describe (reference 7, sections 4.2 and 4.3) allows a process to use R for any number of time units. Bochmann[9] uses an even more complicated *four-cycle signaling* protocol originated by Seitz.[5] Besides including the possibility that R may not be "ready," this protocol also provides a phase for preliminary transfer of information from the selected user to R. Additional complexity in Seitz's protocol results because it is meant for an *asynchronous* arbiter.

Our protocol imposes light constraints on the behavior of a user process. For example, it is perfectly allowable for P_i to request R, wait a while, "become bored," and then do something else; g_i may occur then and be ignored. Another possibility is: After P_i is granted R, it may use R for more than a single time unit and may even never want to release it. The arbiter does nothing to prevent such behavior or guard against its consequences.

In the next section, we define a finite-state automaton A_n that precisely specifies the behavior of a synchronous n-process arbiter. This automaton is a specification that is equivalent to that of Panel 1, but at a *lower level of abstraction.* It is shown in reference 8 that, if the protocol (7) is obeyed, the automaton A_n makes the system in Figure 1 behave according to the high-level temporal specifications.

The Arbiter Automaton

The behavior of the n-process arbiter can be defined by a finite-state automaton A_n. Let Q_n be the set of states of A_n. At any moment, the automaton is in a state that corresponds to some *permutation* of some *subset* of $\{1, \ldots, n\}$. The state specifies *which* processes have requested service, and in what *order* the requests were made. [That storing up to n requests is adequate is a result of the protocol property (7) in Panel 1.] Thus, the number of states of A_n is

$$|Q_n| = \sum_{k=0}^{n} \binom{n}{k} k! = n! \sum_{k=0}^{n} \frac{1}{k} \cong n!\, e$$

if n is moderately large.

State Transitions. If $q(t)$, $r(t)$, and $g(t)$ are the state, input, and output of A_n at (discrete) time t, then the next-state map δ and the output map λ will be such that

$$q(t+1) = \delta(q(t), r(t)), \quad g(t) = \lambda(q(t))$$

An automaton with such an output map is known as a *state-*

output automaton. As we will see later, defining A_n as a state-output automaton makes its implementation easier.

To define the next-state and output maps of A_n, we use the following notation:

- $i_1, i_2, \ldots, i_k$ and $j_1, j_2, \ldots, j_l$ denote *distinct* elements of $1, \ldots, n$ (process identifiers). Further, $\{j'_1, \ldots, j'_l\}$ is $j_1, \ldots, j_l$ arranged in *increasing order*.
- The *state* of A_n is either ϕ or $i_1 \cdots i_k$, for some $k \leq n$. State ϕ means that there are no pending requests for the resource, while state $i_1 \cdots i_k$ means that processes $P_{i1}, \ldots, P_{ik}$—in that order—have requested the resource.
- The *input* to A_n is an element of $\{0, 1\}^n$. For convenience, we use 0 to denote the all-zero input and $r_{j1} \cdots r_{jl}$ for the input that has 1s in positions $j_1, \ldots, j_l$, with $l \leq n$. Input 0 means that nobody is requesting the resource, while input $r_{j1} \cdots r_{jl}$ means that processes $P_{j1}, \ldots, P_{jl}$ simultaneously request the resource.
- Finally, the *output* of A_n is either 0 (nobody is granted the resource) or g_i (P_i is granted the resource).

Panel 2 uses this notation to define the next-state and output maps of A_n. Note that the automaton is only *partially defined*: The constraint on what inputs apply to a state reflects that A_n is designed to operate under the protocol specification (7) in Panel 1. [To be precise, that $i_1, \ldots, i_k$ are distinct is *also* a result of this protocol property. Also, one could design a more *robust* automaton by specifying what happens if $\{i_1, \ldots, i_k\} \cap \{j_1, \ldots, j_l\} \neq \phi$ or $l > n - k$.

Note how the next-state map δ treats simultaneous requests: It gives priority to those from *lower*-numbered processes, something that is *not* required by the specifications in Panel 1. To clarify the operation of A_n even further, Figure 2 shows the state diagram of A_3. Some sequences of state transitions in this diagram are *prohibited* by the protocol specification in Panel 1.

Minimality of A_n. One question arises naturally: Is the specification of A_n in Panel 2 *minimal* (i.e., does it have the least possible number of states)? The structure of A_n is

Panel 2: The automaton A_n

Constraints

$i_1, \ldots, i_k$ and $j_1, \ldots, j_l$ are distinct, $0 \leq k + l \leq n$, and $\{i_1, \ldots, i_k \cap j_1, \ldots, j_l\} = \phi$.

Next-state map δ

$\delta(\phi, 0) = \phi$

$\delta(\phi, r_{j1} \cdots r_{jl}) = j'_1 \cdots j'_l$, where $\{j'_1, \ldots, j'_l\}$ is $\{j_1, \ldots, j_l\}$ inincreasing order

$\delta(i_1 \cdots i_k, 0) = i_2 \cdots i_k$, or ϕ if $k = 1$

$\delta(i_1 \cdots i_k, r_{j1} \cdots r_{jl}) = i_2 \cdots i_k j'_1 \cdots j'_l$

Output map λ

$\lambda(\phi) = 0 \quad \lambda(i_1 \cdots i_k) = g_{i1}$

such that we can easily prove that the automaton is minimal. Indeed, if $A_n[q]$ is the input-output behavior of A_n when started in state q, then one can easily see from Panel 2 that the *null* input 0 has the property

$$\forall q, q' \in Q_n, \quad A_n[q](0^+) \neq A_n[q'](0^+)$$

This property says that no two states of A_n are equivalent (0^+ is any finite sequence of zeros).

The Reachable States of A_n. We saw earlier that Q_n contains about $e\,n!$ states. It turns out that about 60 percent of these states are reachable from the initial state ϕ of A_n. More precisely, if $Q_n^* \subset Q_n$ is the set of states of A_n that are reachable from ϕ, then

Theorem 1

$|Q_n^*| = |Q_n| - (n! - 1)$. For $n \geq 5$, $|Q_n^*| \simeq (e - 1)n!$.

We will use this theorem to prove some statements about realizations of A_n from smaller modules. The proof provides considerable insight into the operation of A_n, but there is no space for it here (see reference 8, section 3.3). We have shown that all states in $Q_n^* = Q_n$

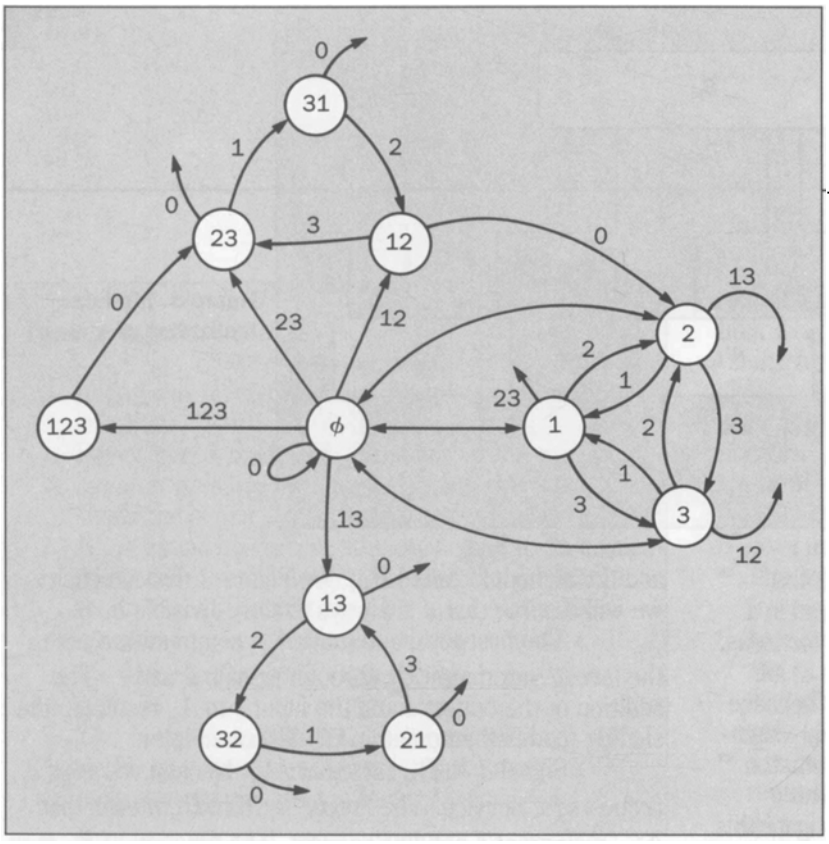

Figure 2. The state diagram of A_3. (The output at a state is the first element of that state.)

$- \{i_1 \cdots i_n \mid i_1 \cdots i_n \neq 1 \cdots n\}$ are reachable from ϕ. Because ϕ is reachable from any state, the set Q_n^* is strongly connected.

Problems with Modular Realizations. The number of states of A_n grows very fast with n, and its reachable states form a strongly connected graph. These facts are evidence that the complexity of a circuit to realize A_n may be impractically high for large n. Because one can always use more time to save space, this statement applies to implementations that must be as fast as possible. As will be seen later, if speed is not important, the automaton A_n can be easily realized using some RAM (random access memory) and a simple arithmetic-logic unit. Sometimes, realizations from smaller modules (Figure 3) can save space without adding much time.

A reasonable first step is to investigate modular realizations of A_n that meet the following restriction: All modules are smaller arbiters chosen from the set $\{A_2, \ldots, A_{n-1}\}$, or *arbitrary* combinational logic modules (one-state automata).

Let n be evenly divisible by m; i.e., $n = mM$. We will consider a subclass of the above class of realizations defined, for $2 \leq m \leq n/2$, as follows: An *m-modular realization of* A_n is an arbitary interconnection of M A_m, one A_m, and arbitary combinational logic modules that defines a state-output automation equivalent to A_n.

We will show that, for $n \geq 7$, m-modular realizations of A_n are *impossible* for any m. This is done by showing that the *number of states* of A_n is *greater* than the number of states of the supposed modular realization. This would not constitute an impossibility proof if A_n were not minimal, but we showed earlier that it is minimal. The following theorem is proved in reference 8:

Theorem 2

If $n \geq 7$, then for any $2 \leq m \leq n/2$,
$| Q_n^* | > | Q_M^* | | Q_m^* |^M$.

The theorem says that straightforward minimal modularizations of the arbiter automaton are impossible, which points out the desirability of a *non*-minimal modularization. (The realization of A_n by A_{m1}, ..., A_{mk} and A_k is "minimal" if $m_1 + m_2 + \cdots + m_k = n$. As defined, an m-modular realization is minimal.)

Theorem 2 is based on the fact that $| Q_n^* |$ grows very fast with n. Another way to prove that a proposed modular realization is not possible is to show that the resulting automaton violates some specification in Panel 1. A good candidate is the precedence requirement (5). However, such a proof may require knowing the combinational logic modules used in the realization.

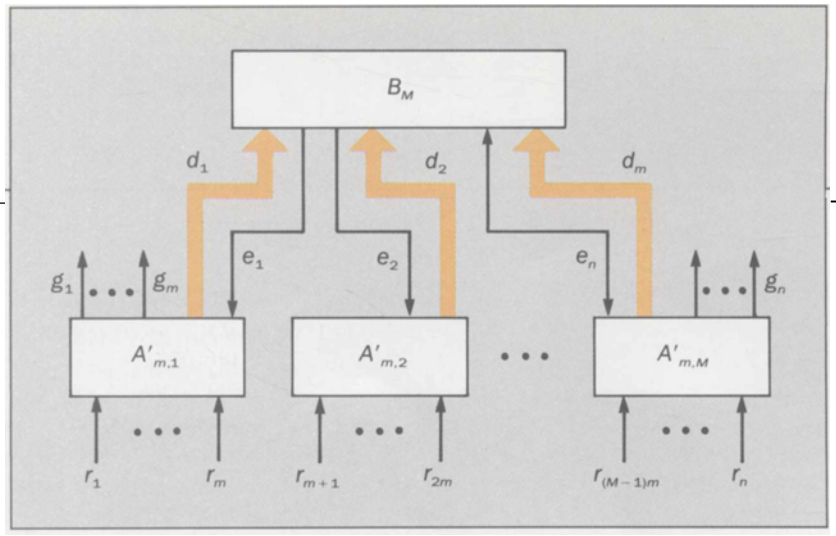

Figure 3. Modular realization of A_n.

From reading this section, it appears that the automaton A_n describes the arbiter as precisely, and in a more intuitively satisfying way than the temporal formulas. Why then bother with the temporal specifications at all?

These specifications are useful precisely because they are at a *higher* level of abstraction than A_n. For example, they leave open the question: Is there an automaton other than A_n with the same external behavior? The answer is *yes*, but we have not attempted to investigate this issue. Besides raising questions, the temporal specifications also provide some answers. For example, it is possible to show rigorously that there is no one-state automaton (combinational circuit) with the desired temporal properties.

Modular Realization of the Arbiter Automaton

Given M identical m-input arbiter modules $A_{m,1}$, $A_{m,2}$, ..., $A_{m,M}$, what do we need to add to make an arbiter A_n that can handle $n = mM$ processes? After the considerations of the previous section on simply adding an M-input arbiter A_M, some thought shows that the difficulty in partitioning A_n into smaller modules lies in satisfying the arbiter's first-come, first-served property. The scheme to be described accomplishes a correct modularization by

- A trivial modification of the automaton A_M, that results in A'_m
- The addition to $A'_{m,1}$, ..., $A'_{m,M}$ of a module B_M that is more complex than A_M.

General Scheme. Suppose we want to build an A_n that uses several A_m, where $m \geq 2$. Letting $M = n/m$, we will use M A_m—denoted $A_{m,1}$, $A_{m,2}$, ..., $A_{m,M}$—and an additional module called B_M; see Figure 3. For simplicity, we will assume that n is always exactly divisible by m.

The first-level automata $A_{m,i}$ communicate with the level-2 automaton B_M through signals d_i and e_i. The addition of the output d and the input e to A_m results in the slightly modified automaton A'_m described later.

Signal d_i to B_M says that $A'_{m,i}$ has just received d_i requests for service. The "reply" e_i from B_M means that $A'_{m,i}$ may grant a pending request. The function of B_M is to ensure that, if a request r_j arrives at $A'_{m,i1}$ before some other request r_k arrives at $A'_{m,i2}$, then $A'_{m,i1}$ grants r_j before $A'_{m,i2}$ grants r_k. To do this, B_M enables the level-1 automata at the appropriate times. A little thought shows that if B_M were simply A_M, this function would not be performed correctly.

Algorithm $\mathcal{A}$ (Panel 3) describes more precisely, although informally, how the system of Figure 3 works. The basic idea is that B_M maintains a *queue* of *pairs* of the form (i, d_i), where i identifies a level-1 module, and d_i is the number of requests received by that module at some past instant. This queue is the state of B_M.

Notice that B_M behaves like an automaton A_M, except it has a *two-part* state. To understand the system better, consider the case where, at some instant t, the queue of B_M contains $(3, c_3)$ $(1, c_1)$ $(7, c_7)$ $(1, c'_1)$. Independently of what inputs arrive at the level-1 modules for time $\geq t$, $A'_{m,3}$ will be enabled for c_3 time units after t. $A'_{m,1}$ will be enabled next for c_1 time units, and $A'_{m,7}$ after it for c_7 time units. Finally, $A'_{m,1}$ will be enabled again for c'_1 time units. So, we can think of the output d_i of $A'_{m,i}$ as a request to be enabled for *duration* d_i.

Panel 3: Algorithm $\mathcal{A}$

0. Initially, all level-1 modules are in state ϕ, and B_M is in state $(\phi, 0)$.
1. Every level-1 module $A'_{m,i}$ reads its input $r_{j1} \cdots r_{jl}$, modifies its state by appending $j_1 \cdots j_l$ to it, and sends the signal $d_i = l$ to B_M.
2. B_M modifies its current state by adding to its queue the nonzero requests d_i of the level-1 modules. It then looks at the first element (j, c_j) of the queue, enables $A'_{m,j}$ by signaling e_j, and decrements c_j. If the result is 0, then $(j, 0)$ is removed from the queue.
3. The (unique) enabled level-1 automaton $A'_{m,j}$ generates its output g and completes the move to its next state as specified in Panel 2.

The following issues arise about algorithm $\mathcal{A}$:

- Prove that the algorithm is *correct*; i.e., prove that the automaton $\mathcal{A}_n$ it defines (after some details are specified) behaves exactly like A_n. The proof of this may be found in reference 8.
- Formally specify the automaton A'_m. This is easy (see the next section).
- Formally specify the level-2 automaton B_M. In particular, put a bound on the number of its states (maximum queue length). Clearly, the length of the queue maintained by B_M can never exceed n as there are only n processes in the system, and the protocol specifications in Panel 1 forbid repeated requests. We will see that a queue length of $N = n - 1$ is sufficient (and also necessary).

The Level-1 Automaton A'_m. The external behavior of A'_m is identical to that of A_m. Internally, however, its next-state map δ operates in *two phases* according to the input e, and there is an additional component λ' of the output map that depends only on the input and produces d. The map δ of A'_m is defined as the composition $\delta_s \circ \delta_a$ of two component maps δ_s and δ_a. The map δ_a is active when $e = 0$ (phase 1 of algorithm $\mathcal{A}$) and performs an "append" function. The map δ_s is active when $e = 1$ and performs a "shift" function (phase 3 of $\mathcal{A}$).

In Panel 4, which defines the automaton A'_m, we use the notation introduced in Panel 2. Notice the convention that, if $\mu = 0$, then $i_1 \cdots i_\mu$, $i_2 \cdots i_\mu$, and $j'_1 \cdots j'_\mu$ are all ϕ, while $r_{j1} \cdots r_{j\mu}$ is zero.

The Level-2 Automaton B_M. To describe the automaton B_M, we use notation analogous to that for A_n:

- The *state* of B_M consists of two parts and is denoted $(i_1 \cdots i_k, c_1 \cdots c_k)$. $i_1, \ldots, i_k$ are level-1 automaton identifiers (not necessarily distinct elements of $\{1, \ldots, M\}$), and $c_1, \ldots, c_k$ are request counts (in $\{1, \ldots, m\}$). The first part of the state may be ϕ, and the second may be zero.
- The *input* to B_M is either zero (no level-1 automaton is requesting service), or $d_{j1} \cdots d_{jl}$. In this case, automata $A'_{j1}, \ldots, A'_{jl}$ are simultaneously requesting service of duration $d_{j1}, \ldots, d_{jl}$, respectively. (B_M knows which level-1 automata are requesting service, because $A'_{m,1}$ is connected to d_1; $A'_{m,2}$ is connected to d_2; etc.)
- The *output* of B_M is either zero (no level-1 automaton is enabled), or e_i (automaton $A'_{m,i}$ is enabled).

Panel 5 defines the next-state and output maps of B_M, using this notation. It also uses the convention that $i_2 \cdots i_k = \phi$ and $c_2 \cdots c_k = 0$, if $k = 1$. Notice that the next-state map δ of B_M is divided into components δ_I and δ_C.

The *identifier part* δ_I maps the i-part of the state and input into the i-part of the state. (δ_I also depends on c_1 of the state. To simplify the notation, we have made the action of δ_I conditional on the value of c_1.) The *counter* part δ_C maps the c-part of the state and input into the c-part of the state.

Finally, B_M is a state-output automaton like A_n, and its output map λ depends only on the c-part of the state.

Bounds on Number of Reachable States. The *implementation* of the level-2 automaton B_M that was described earlier used a queue of length $N \leq n$ (n was evidently sufficient). Theorem 3, proved in reference 8, shows that

- All reachable states of B_M have no more than $n - 1$ elements, so a queue length of $n - 1$ is *adequate*.
- Under certain conditions, some states with $n - 1$ elements are indeed reachable, so a queue length of $n - 1$ is also *necessary*.

Theorem 3

All states of B_M that are reachable from the initial state $(\phi, 0)$ have at most $n - 1$ elements, and there is at least one state with $n - 1$ elements that is reachable from $(\phi, 0)$.

As we will see later, this theorem unfortunately implies that it takes about as many logic gates to implement B_M as it takes to implement A'_n. This means that the modular realization doesn't save us anything! However, suppose that the users' behavior is such that, with high probability, the queue length N of B_M does not exceed $cM^{1+\epsilon}$, where c and ϵ are some constants. Then, we will show that the modular automaton $\mathcal{A}_n$ can be realized at *much lower cost* than A_n. As long as the queue of B_M never overflows, this automaton will remain equivalent to A_n. That is, the modular realization of the arbiter will behave just like a "monolithic" one.

Panel 4: The automaton A'_m

Constraints

As in Panel 2

Next-state map $\delta = \delta_s \circ \delta_a$

$$\delta_a(i_1 \cdots i_k, r_{j1} \cdots r_{jl}) = i_1 \cdots i_k j'_1 \cdots j'_l$$

$$\delta_s(i_1 \cdots i_\mu) = i_2 \cdots i_\mu$$

Output map

$$\lambda(i_1 \cdots i_k) = \begin{cases} 0, & e = 0 \\ g_{i1}, & e = 1 \end{cases}$$

$$\lambda'(r_{j1} \cdots r_{jl}) = \begin{cases} l, & e = 0 \\ 0, & e = 1 \end{cases}$$

Implementing a Level-1 Module

Here we find fairly abstract (but constructive) upper bounds to the *size* and *speed* of an implementation of the automaton A'_m. We interpret "size" in the gate-count sense, and not in the (less fundamental) layout-area sense used in VLSI.

Let $\mathbf{C}(A'_m)$ be the minimum possible number of logic gates needed to implement A'_m, and let $\mathbf{D}(A'_m)$ be the speed of the fastest possible circuit that realizes A'_m. We will call $\mathbf{C}(A'_m)$ the *complexity* and $\mathbf{D}(A'_m)$ the *delay* of A'_m. We will show by construction that $\mathbf{C}(A'_m) \leq O(m^2 \log m)$, while $\mathbf{D}(A'_m) \leq O(\log m)$. We will also show that our bound on the delay is the best possible.

As mentioned earlier, our implementation will be synchronous, which requires that the inputs $r_1, \ldots, r_m$ to A'_m be synchronized with the automaton's clock. The synchronization problem is subtle and not dealt with here. Suffice it to say that, under very mild constraints, a reliable synchronizer (perfectly bistable device) is known to be theoretically impossible.[6] Techniques for achieving reliable synchronization in practice are analyzed in reference 10.

Besides assuming that the inputs to A_m are already synchronized, we will also ignore certain other practical issues when describing our implementation, notably fan-out limitations of logic elements.

Complexity and Delay. The *complexity* of an m-input, n-output Boolean function ϕ is the minimum number of gates needed to construct a circuit that computes ϕ. The gates are selected from a set of gates called a *basis*. We will generally use the basis {*AND, OR, NOT, XOR*}. The *delay* of ϕ is that of the fastest circuit that realizes ϕ. If the basis is changed, the complexity and delay may change. (For details on these concepts, see reference 11.)

We will use the following notation:

- AND_n will denote an n-input AND gate, and OR_n an n-input OR gate, while f will indicate the fan-in of the gates in the basis. Fan-out limitations will be ignored.
- "lg" indicates a base-2 logarithm, and "Lg" is the ceiling of lg. Also, Log_f is the ceiling of $\log_f$.

For the gate complexities, we will assume that

Panel 5: Next-State and Output Maps of Automaton B_M

"Identifier" map δ_I

$$\delta_I(\phi, 0) = \phi$$

$$\delta_I(\phi, d_{j1} \cdots d_{jl}) = j'_1 \cdots j'_l$$

$$\delta_I(i_1 \cdots i_k, 0) = \begin{cases} i_1 \cdots i_k, & c_1 > 1 \\ i_2 \cdots i_k, & c_1 = 1 \end{cases}$$

$$\delta_I(i_1 \cdots i_k, d_{j1} \cdots d_{jl}) = \begin{cases} i_1 \cdots i_k j'_1 \cdots j'_l, & c_1 > 1 \\ i_2 \cdots i_k j'_1 \cdots j'_l, & c_1 = 1 \end{cases}$$

"Counter" map δ_C

$$\delta_C(0, 0) = 0$$

$$\delta_C(0, d_{j1} \cdots d_{jl}) = d_{j'1} \cdots d_{j'l}$$

$$\delta_C(c_1 \cdots c_k, 0) = \begin{cases} (c_1 - 1)\, c_2 \cdots c_k, & c_1 > 1 \\ c_2 \cdots c_k, & c_1 = 1 \end{cases}$$

$$\delta_C(c_1 \cdots c_k, d_{j1} \cdots d_{jl}) = \begin{cases} (c_1 - 1) c_2 \cdots c_k d_{j'1} \cdots d_{j'l}, & c_1 > 1 \\ c_2 \cdots c_k\, d_{j'1} \cdots d_{j'l}, & c_1 = 1 \end{cases}$$

Output map λ

$$\lambda(\phi) = 0$$

$$\lambda(i_1 \cdots i_k) = e_{i1}$$

$$\mathsf{C}(AND_n) = \mathsf{C}(OR_n) = \alpha n, \quad \mathsf{C}(XOR_2) = 4\alpha$$

These assumptions are appropriate for MOS (metal-oxide-semiconductor) technology and probably conservative for bipolar technology.

For the gate delays, we will assume that

$$\mathsf{D}(AND_f) = \mathsf{D}(OR_f) = \tau(f), \quad \mathsf{D}(XOR_2) = 2\tau(2)$$
$$\mathsf{D}(AND_n; f) = \mathsf{D}(OR_n; f) = \tau(f) \operatorname{Log}_f n$$

where $\mathsf{D}(AND_n; f)$ is the delay of an AND_n that is built from AND_f. The function $\tau(f)$ depends on the technology used:

$$\tau(f) = \begin{cases} \tau_m f & \text{for MOS} \\ \tau_b & \text{for bipolar} \end{cases}$$

We state without proof that the complexity and delay of an n-to-2^n decoder built with gates of fan-in f are

$$\mathsf{C}(dec(n); f) \leq 2^n(1 + \epsilon)\, af,$$
$$\mathsf{D}(dec(n); f) = \tau(f) \operatorname{Log}_f n + \tau(2) \qquad (1)$$

where $\epsilon = 2(f + 1)2^{-n(1 - 1/f)}$.

Use of a Queue. Our implementation of A'_m will be based on an m-stage queue Q. (There is no relationship between this Q and the Qs that denote state sets in preceding sections.)

The use of the queue is strongly suggested by the form of the next-state map δ of the automaton and has the following advantages:

- The dependence of δ on the present state is greatly reduced.
- Given the queue Q, the component δ_s of δ can be implemented at little additional cost.
- The *state-assignment* (state encoding) problem for the automaton disappears.
- Almost no logic is needed to realize the output map λ of A'_m.
- The *correctness* of the entire implementation needs little argument.

Each stage of the queue Q holds a number $\in \{0, 1, \ldots, m\}$. If the number is zero, the stage is considered to be empty; otherwise, it is full. The queue may be either *decoded* (a stage consists of m flip-flops) or *encoded* [only $\operatorname{Lg}(m + 1)$ flip-flops are needed].

Theorem 1 says that, when m is large, at least

$(m + 1/2)\lg m - m/\ln 2$ flip-flops are needed; in this sense, the encoded queue is asymptotically optimal.

Realization of the map δ_a. The map δ_a has components $q_1, \ldots, q_k, \ldots, q_m$, where q_k controls the kth stage of the queue. Component q_k is a function of $r = (r_1, \ldots, r_m)$ and of $\mu_s \in \{0, 1, \ldots, m,$ which marks the *last full stage* of the queue immediately after the last shift by δ_s.

Proceeding top-down, we note that the function $q_k(r, \mu_s)$ takes the value $i \in \{1, \ldots, m\}$ if and only if $r_i = 1$ and one of the following conditions holds:

$$\begin{aligned} &\mu_s = 0, \text{ and } (r_1, \ldots, r_{i-1}) \text{ contains } k - 1 \text{ ones} \\ &\mu_s = 1, \text{ and } (r_1, \ldots, r_{i-1}) \text{ contains } k - 1 \text{ ones} \\ &\qquad\vdots \\ &\mu_s = k - 1, \text{ and } (r_1, \ldots, r_{i-1}) \text{ contains 0 ones} \end{aligned} \tag{2}$$

If none of these conditions is true, then $q_k(r, \mu_s) = 0$.

Now, let the function $c_i(r_1, \ldots, r_i)$ compute the number of 1s in the vector $(r_1, \ldots, r_i)$. Then,

$$q_k(r, \mu_s) = i \Leftrightarrow (r_i = 1) \wedge \bigvee_{j=0}^{k-1} (\mu_s = j) \wedge (c_{i-1} = k - j - 1) \tag{3}$$

where $\wedge$ and $\vee$ denote logical AND and logical OR.

We would like to find out if δ_a can be realized more economically with a decoded queue or with an encoded queue. If the queue is decoded, q_k consists of m subfunctions that are easily synthesized from equation (3). As shown in reference 8, this leads to the estimate $C(\delta_a) = O(m^3)$. Intuitively, using an encoded queue should result in a less complex (but maybe not faster) realization of δ_a. Unfortunately, it is not clear how to synthesize the $\mathrm{Lg}(m + 1)$ components of q_k based on equation (3) when the queue is encoded.

To resolve the difficulty, we retrace some of our steps, to discover that $q_k(r, \mu_s) = i$ if and only if $r_i = 1$ and

$$\text{the number of 1s in } (r_1, \ldots, r_{i-1}) \text{ plus } \mu_s \text{ equals } k - 1 \tag{4}$$

This condition, which is equivalent to equation (2) and more useful, is easy to see when one thinks of the map δ_a as a *switching network* of the connection type (see, for example, reference 12). But it seems difficult to arrive at equation (4) by proceeding in a strictly top-down fashion. Using the counting functions c_i, we now have

$$q_k(r, \mu_s) = i \Leftrightarrow (r_i = 1) \wedge (\mu_s + c_{i-1} = k - 1) \tag{5}$$

which says that δ_a maps input i to output k if and only if the sum of the marker μ_s and the number of 1s in $r_1, \ldots, r_{i-1}$ is $k - 1$. The shift to the connection-network viewpoint seems crucial, because now the realization of δ_a using *either* type of queue is straightforward and, clearly, the encoded queue leads to a more economical realization.

Implementation of the Queue and Map δ_s. Because each stage of the queue can contain a number that ranges from zero to m, it must consist of $\mathrm{Lg}(m + 1)$ flip-flops. The whole queue is then an $m \times \mathrm{Lg}(m + 1)$ array of flip-flops. By choosing $R - S$ flip-flops, the "shifting" map δ_s is trivially realized by the logic shown in Figure 4. Note that the logic is such that the "append" operation can write only 1s into a flip-flop.

If $C(R - S \text{ ff}) \leq 4\alpha$ and $D(R - S \text{ ff}) \leq 2\tau(2)$, we find that

$$\begin{aligned} &C(Q, \delta_s) \leq 13\alpha m \mathrm{Lg}(m + 1) + C(M_s), \\ &D(Q, \delta_s) \leq 4\tau(2) \end{aligned} \tag{6}$$

where $C(M_s)$ is given by equation (22).

Detailed Realization of δ_a. By what was said in the two preceding subsections, the mapping performed by δ_a can be realized as follows. First, the functions $c_1(r_1), \ldots, c_{m-1}(r_1, \ldots, r_{m-1})$ are computed. Then, a set of m adders is used to compute

$$a_k = c_{k-1} + \mu_s + 1 \tag{7}$$

for $k = 1, \ldots, m$; for $k = 1$, define c_0 to be zero. The a_k

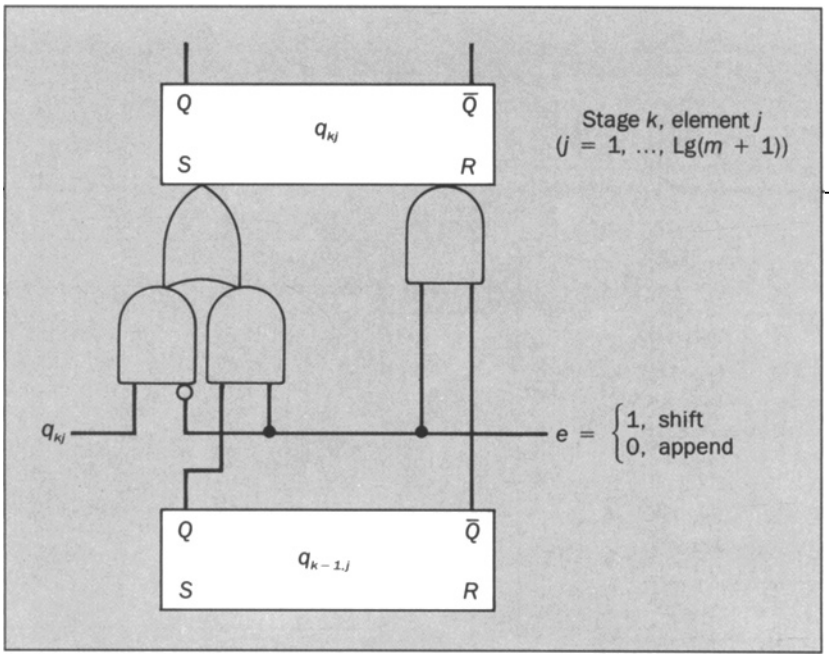

Figure 4. Element q_{kj} of the queue Q.

are then used as *addresses* for an m-input, m-output *switch* with Lg(m + 1) bit-wide paths, which produces q_1, ..., q_m. Finally, the value c_m is added to μ_s to produce the marker μ_a, which identifies the last full stage of the queue immediately after δ_a does the "append" operation. This architecture, augmented by a few details, can be seen in Figure 5.

We now describe some parts of Figure 5 in more detail.

The block R. This block provides the *data* to the switch. Its function is defined by

$$R_i = \begin{cases} 0, \text{ if } r_i = 0 \\ \text{the binary representation of } i, \text{ if } r_i \neq 0 \end{cases}$$

Each R_i is implemented by Lgm AND_2s, so

$$\mathsf{C}(R) \leq 2\alpha m \text{Lg} m, \quad \mathsf{D}(R) \leq \tau(2) \tag{8}$$

The switch S. The switch $S_{m,m,\text{Lg}(m+1)}$ consists of Lg(m + 1) identical copies of an $m \times m$ switch $S_{m,m,1}$. If x_1, ..., x_m and y_1, ..., y_m are the inputs and outputs of $S_{m,m,1}$, this switch can realize *any single-valued mapping* from x_1, ..., x_m to y_1, ..., y_m. The mapping is specified by the Lgm-bit addresses a_1, ..., a_m; and a_k specifies the y to which x_k is mapped. Figure 6 shows the implementation of $S_{3,3,1}$.

Because $S_{m,m,1}$ can be built with m^2 AND_2s and m OR_ms

$$\mathsf{C}(S_{m,m,\text{Lg}(m+1)}) \leq 3\alpha m^2 \text{Lg}(m+1),$$
$$\mathsf{D}(S_{m,m,\text{Lg}(m+1)}) \leq \tau(f) \text{Log}_f m + \tau(2) \tag{9}$$

The block D of mLg(m + 1)-to-$2^{\text{Lg}(m+1)}$ decoders is characterized by [recall equation (1)]

$$\mathsf{C}(D) \leq m2^{\text{Lg}(m+1)}(1+\epsilon)\alpha f,$$
$$\mathsf{D}(D) \leq \tau(f) \text{Log}_f \text{Lg}(m+1) + \tau(2) \tag{10}$$

The block C_m. If the function $c_k(r_1, \ldots, r_k)$ counts the number of 1s in r_1, ..., r_k, then the function C_m computes the cumulative number of 1s in r_1, ..., r_m. C_m is defined as $(c_1(r_1), c_2(r_1, r_2), \ldots, c_m(r_1, \ldots, r_m))$. From what was said earlier, the function λ' is identical to c_m.

The realization of the system of functions C_m is based on a realization of the function c_m, as Figure 7 shows (partially) for m = 16. c_m itself is realized iteratively, based on the following idea: If m is even, to count the number of 1s in r_1, ..., r_m, count the number of 1s in r_1, ..., $r_{m/2}$ and in $r_{m/2+1}$, ..., r_m, and add the results (see reference 11, section 3.1.1).

In implementing c_m, it is convenient to assume that m is a power of two. If it isn't, we will build a circuit that realizes c_m for the next higher power of two and use it by setting the excess inputs to zero.

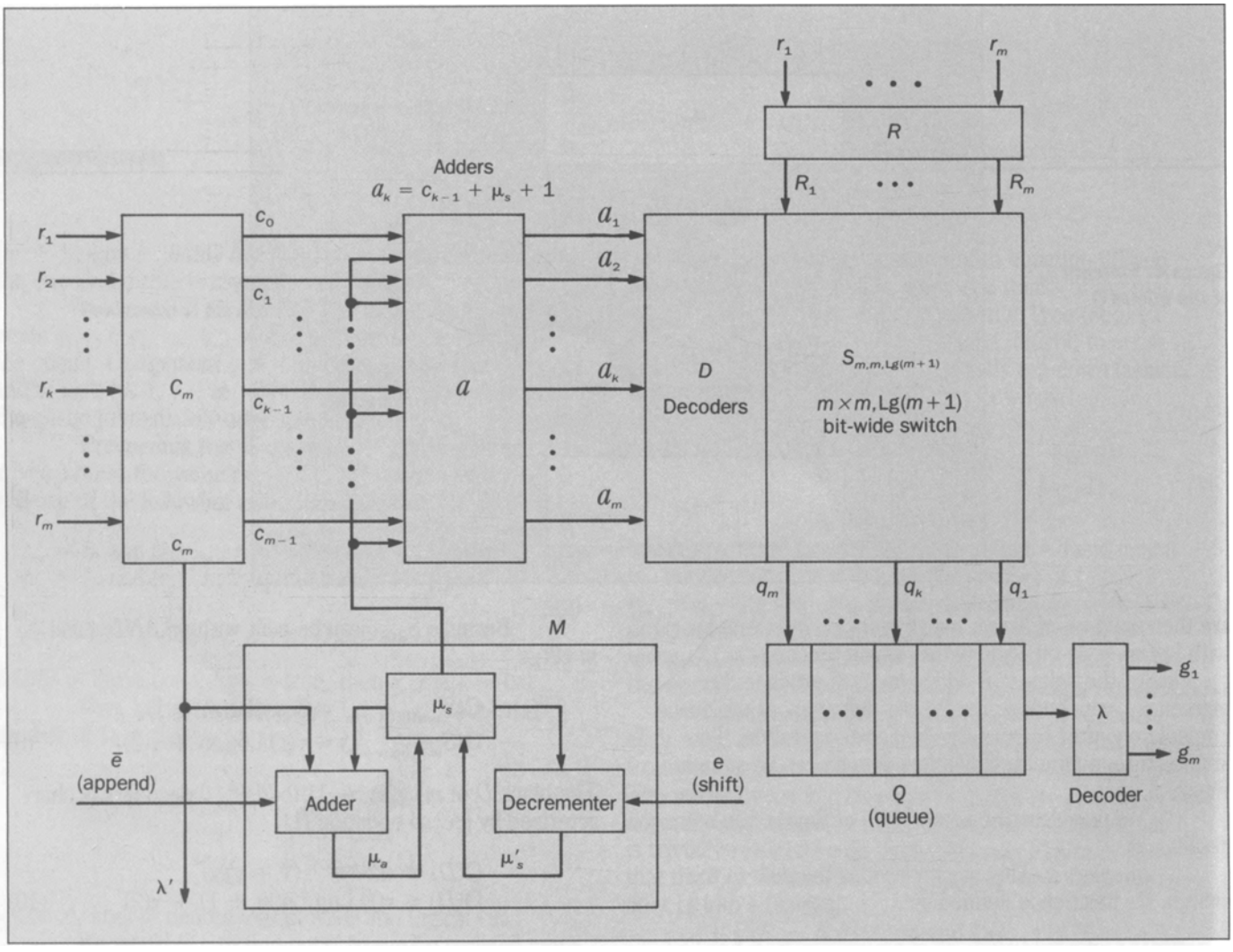

Figure 5. The level-1 automaton A'_m.

The basic component of the implementation is an adder $a_{k,k,k+1}$ with two k-bit inputs and a $(k + 1)$-bit output. If we let L abbreviate Lgm, we observe that c_m is realized by an L-level tree of adders that has 2^{L-k} adders $a_{k,k,k+1}$ at level k. Consequently,

$$\mathbf{C}(c_m) \leq \sum_{k=1}^{L} 2^{L-k}\,\mathbf{C}(a_{k,k,k+1}) \qquad (11)$$

Clearly, this tree for c_m also realizes all functions c_k with k a power of two and $\leq 2^L$. Any other c_k with even index—i.e., $c_{2\ell}$—is constructed inductively, by adding two of the already constructed c_2, c_4, ..., $c_{2(\ell-1)}$. Thus, we can see that:

- To realize all functions c_ℓ, where ℓ is an even number, not a power of two, and $< 2^{L-1}$, we need

$$\sum_{k=3}^{L-1} (2^{k-2} - 1)\,\mathbf{C}(a_{k,k,k+1}) \qquad (12)$$

All these functions depend only on r_1, ..., $r_{m/2}$ and lie in the left half of the tree in Figure 7. Suppose that we also realize exactly the same functions, but the inputs now are $r_{m/2+1}$, ..., r_m. The complexity of the realization

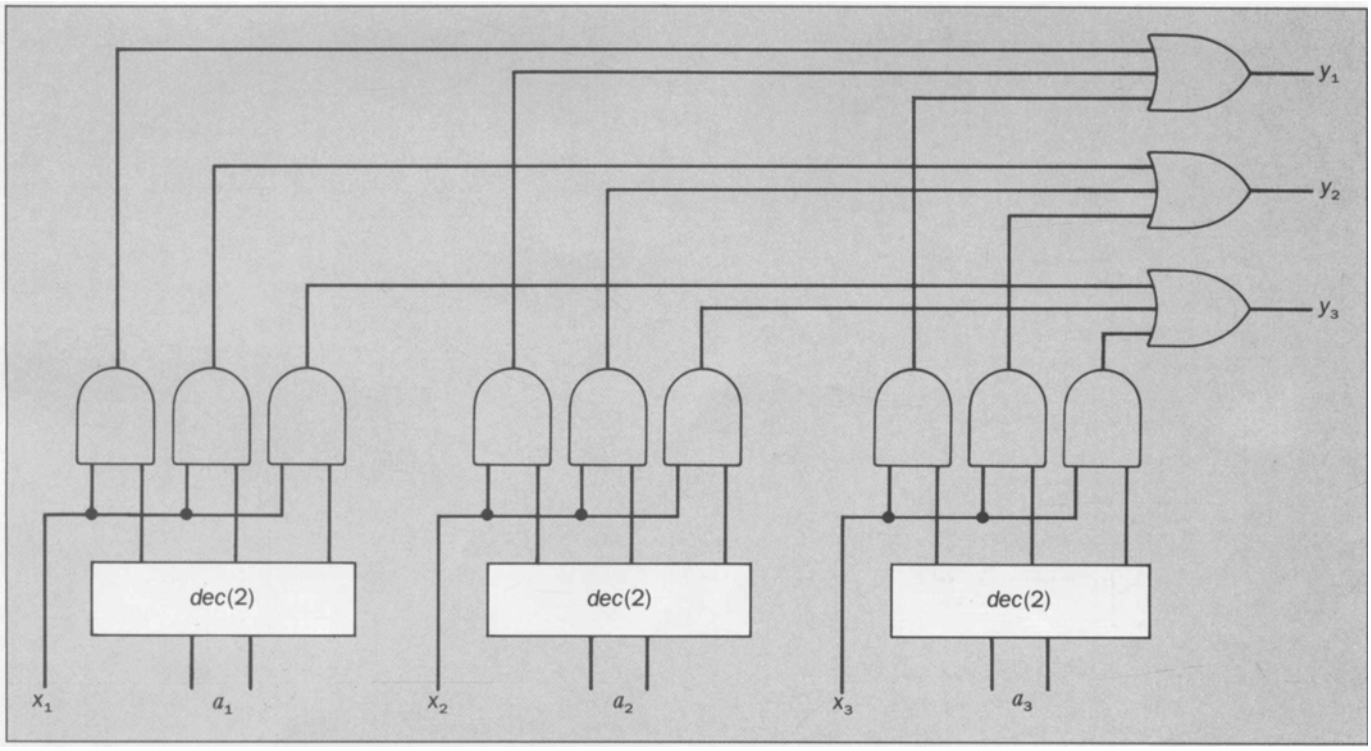

Figure 6. The switch $S_{3,3,1}$ and associated decoders.

is again given by equation (12). Given these auxiliary functions,

- To realize all functions c_ℓ, where ℓ is an even number, not a power of two, and between 2^{L-1} and 2^L, we need

$$(2^{L-2} - 1)\mathsf{C}(a_{L,L,L+1}) \tag{13}$$

(one adder per c_ℓ; and one input of all these adders is $c_{2^{L-1}}$). Finally, to realize cs with odd indexes, notice that $c_{2\ell+1}$ can be obtained by using a copy of the adder that forms $c_{2\ell}$ (i.e., an adder with the same inputs) and setting its carry-in to $r_{2\ell+1}$. Thus, $c_{2\ell+1}$ is formed at the same level as Figure 7 with $c_{2\ell}$. Consequently,

- To realize all functions $c_{2\ell+1}$, we need

$$\sum_{k=1}^{L-1} \mathsf{C}(a_{L,L,L+1}) + (12) + (13) \tag{14}$$

(The first sum corresponds to the even numbers that are powers of two $\leq 2^{L-1}$.)

We conclude from the above that

$$\mathsf{C}(C_m) \leq (11) + 2\cdot(12) + (13) + (14) \tag{15}$$

We will implement $a_{k,k,k+1}$ as a k-stage *serial* adder, and each stage will be the full adder $a_{1,1,2}$ shown in Figure 8. (See the remark at the end of this section.)

As stated earlier, the complexity and delay of this adder are

$$\begin{aligned} &\mathsf{C}(a_{1,1,2}) \leq 28\alpha, \\ &\mathsf{D}(s_i) = d_s = \tau(f)\,(\mathrm{Log}_f 4 + \mathrm{Log}_f 3) + \tau(2), \\ &\mathsf{D}(c_i) = d_c = \tau(f)\,\mathrm{Log}_f 3 + \tau(2) \end{aligned} \tag{16}$$

So, from equation (15),

$$\mathsf{C}(C_m) \leq 28\alpha \left[2^{\mathrm{Lg}m} \left(\frac{5}{4}\,\mathrm{Lg}m + 1\right) - (\mathrm{Lg}m)^2 - 2\mathrm{Lg}m + 1) \right] \tag{17}$$

The delay of C_m does not exceed that of c_m. It can be shown that the ith sum digit and ith carry digit of an adder at level k of the tree that realizes c_m are produced at times

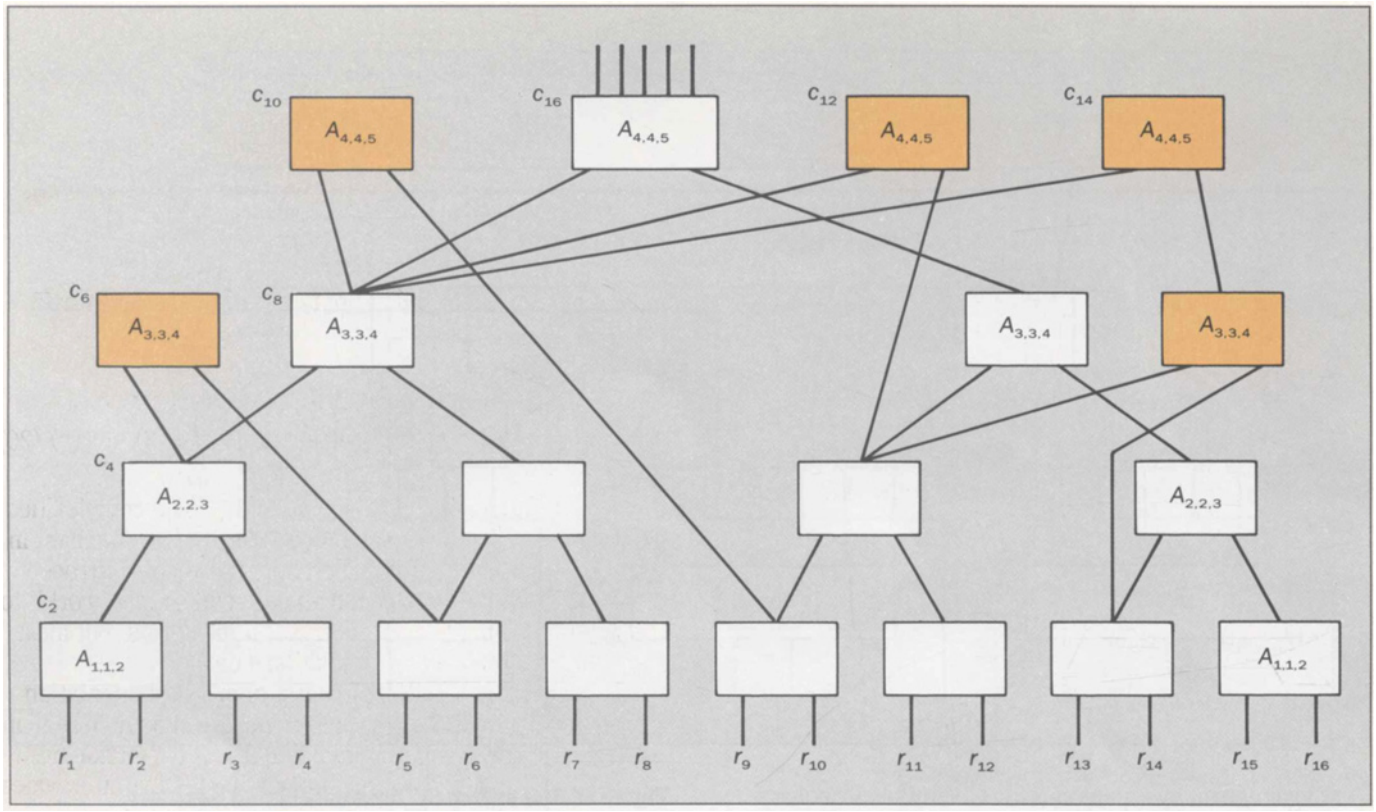

Figure 7. The functions of C_{16} with even indexes.

$$t(s_i) = kd_s + id_c,$$
$$t(c_i) = (k-1)d_s + (i+1)d_c \qquad (18)$$

for $i = 0, \ldots, k-1$. We can derive equation (18) by double induction on k and i based on the fact that

$$t(s_i) = \max\{t_i, t(c_{i-1})\} + d_s,$$
$$t(c_i) = \max\{t_i, t_i(c_{i-1})\} + d_c \qquad (19)$$

where t_i is the time at which the ith pair of input bits is available. From equation (16), $d_s > d_c$, so

$$\mathsf{D}(C_m) \leq (d_s + d_c)\mathrm{Lg}m - d_c \qquad (20)$$

Remark on parallel adders. Using some type of parallel (carry-lookahead) adder to realize c_m does not seem to help reduce the delay. If all inputs to $\mathcal{A}_{k,k,k+1}$ arrive simultaneously, a parallel realization is certainly faster than a serial one. But suppose the inputs are $(a_0, \ldots, a_{k-1})$ and $(b_0, \ldots, b_{k-1})$; and a_0, b_0 arrive at time t_0; a_1, b_1 arrive at t_1; ...; etc. Then, it can be shown that there exist $(t_0, t_1, \ldots, t_{k-1})$ such that (at least a straightforward) parallel realization is *slower* than the serial one.

The blocks $\mathcal{A}$ and M. The adder block $\mathcal{A}$ in Figure 5 produces the functions $a_1, \ldots, a_m$ defined by equation (7). The largest adder is $\mathcal{A}_{\mathrm{Lg}m+1,\ \mathrm{Lg}m+1,\ \mathrm{Lg}m+2}$, needed for a_m (however, only Lgm bits of its output are used). So,

$$\mathsf{C}(\mathcal{A}) < m\,\mathsf{C}(\mathcal{A}_{\mathrm{Lg}m+1,\ \mathrm{Lg}m+1,\ \mathrm{Lg}m+2}) \leq 28\alpha m\mathrm{Lg}(m+1) \qquad (21)$$

$$\mathsf{D}(\mathcal{A}) \leq \mathsf{D}(a_m) \leq \text{by (19)} \leq d_s \qquad (22)$$

Block M in Figure 5—which consists of a register, an adder, and a decrementer—realizes the function μ whose value $\in \{0, 1, \ldots, m\}$ indicates the last full stage of the queue. The adder that produces $\mu_a = \mu_s + c_m$ is a Lg$(m+1)$-stage ripple adder with delay d_s. The decrementer Δ is implemented in a parallel fashion, and we state without proof that

$$C(\Delta) \leq \left[\frac{1}{2}(\mathrm{Lg}(m+1))^2 + 9\mathrm{Lg}(m+1)\right]\alpha$$

$$D(\Delta; f) \leq \tau(f)\,\mathrm{Log}_f \mathrm{Lg}(m+1) + 3\tau(2)$$

We will take the block M to consist of the subblocks M_a and M_s that are considered parts of δ_a and δ_s, respectively. Thus,

$$C(M_a) \leq 32\alpha \mathrm{Lg}(m+1), \quad C(M_s) \leq C(\Delta) \tag{23}$$

$$D(M_a) \leq d_s + 2\tau(2), \quad D(M_s) \leq D(\Delta; f) + 2\tau(2) \tag{24}$$

where the $2\tau(2)$ is the delay of the register that holds μ_s.

Complexity and Delay of A'_m. Clearly, $C(A'_m) \leq C(\delta_a, \lambda') + C(Q, \delta_s) + C(\lambda)$, where from Figure 5,

$$C(\delta_a, \lambda') \leq C(C_m) + C(a) + C(D) + C(S) + C(R) + C(M_a)$$

The right-hand side can be computed from equations (17), (21), (10), (9), (8), and (23). $C(Q, \delta_s)$ is given by equation (6), and $C(\lambda) = C(dec(\mathrm{Lg}(m+1)))$. We thus find that

$$C(A'_m) \leq (3m^2 \lg m + (3 + 2(1+\epsilon)f)m^2 + 113m \lg m + (197 + 2(1+\epsilon)f)m - 21.5(\lg m)^2 - 2\lg m + 75.5)\,\alpha \tag{25}$$

where we used $\mathrm{Lg}(m+1) \leq \lg m + 1$. Concerning the delay of A'_m, we observe that $D(A'_m) \leq D(\delta_a) + D(\delta_s) + D(\lambda)$, where

$$D(\delta_a) \leq D(C_m) + D(a) + D(D) + D(S) + D(Q, \delta_s)$$
$$D(\delta_s) \leq \max\{D(Q, \delta_s), D(M_s)\}$$
$$D(\lambda) \leq D(dec(\mathrm{Lg}(m+1)))$$

We can find $D(\delta_a)$ from equations (20), (22), (10), (9), and (6); $D(\delta_s)$ from equations (6) and (22); and $D(\lambda)$ from equation (1). If $f \geq 4$ to simplify equation (16), we finally find

$$D(A'_m) \leq (3\tau(f) + 2\tau(2))\mathrm{Lg}m + \tau(f)\,\mathrm{Log}_f m + 3\tau(f)\,\mathrm{Log}_f \mathrm{Lg}(m+1) + 12\tau(2) - \tau(f) \tag{26}$$

Comments on the Realization of A'_m. The complexities of the switch S and associated decoders D are dominant in our implementation of A'_m. Perhaps, these complexities could be reduced by using a cheaper switching network[12] to realize the "append" operation that δ_a performs, but most probably at the expense of additional delay.

What dominates the delay of our implementation is the delay of block C_m, which, in turn, can be no less than the delay of the 1-counting function c_m. This function has many interesting properties; among them is that it provides the basis for the economical realization of any *symmetric* Boolean function (see reference 11, section 3.1.2). Clearly, the least-significant bit of c_m is $r_1 \oplus \cdots \oplus r_m$, which is the *parity* function on $r_1, \ldots, r_m$. It follows from Savage's Theorems 3.3.1.3 and 2.3.3[11] that

Lemma 1

Over the basis $\{AND_2, OR_2, NOT\}$, c_m cannot be realized with delay less than or equal to $2\tau(2)\lg m$.

But if gates of *unbounded* fan-in are available, any Boolean function can be realized in two levels of logic (excluding inversions). c_m is "bad" in that respect, because by Theorem 2.1.2 on the parity function[11]

Lemma 2

Even with unlimited fan-in, to realize c_m with depth two requires a number of gates that are exponential in m.

From Furst, Saxe, and Sipser's results[13] on the parity function, it also follows that if we try to realize c_m with depth equal to any constant $d > 2$, then

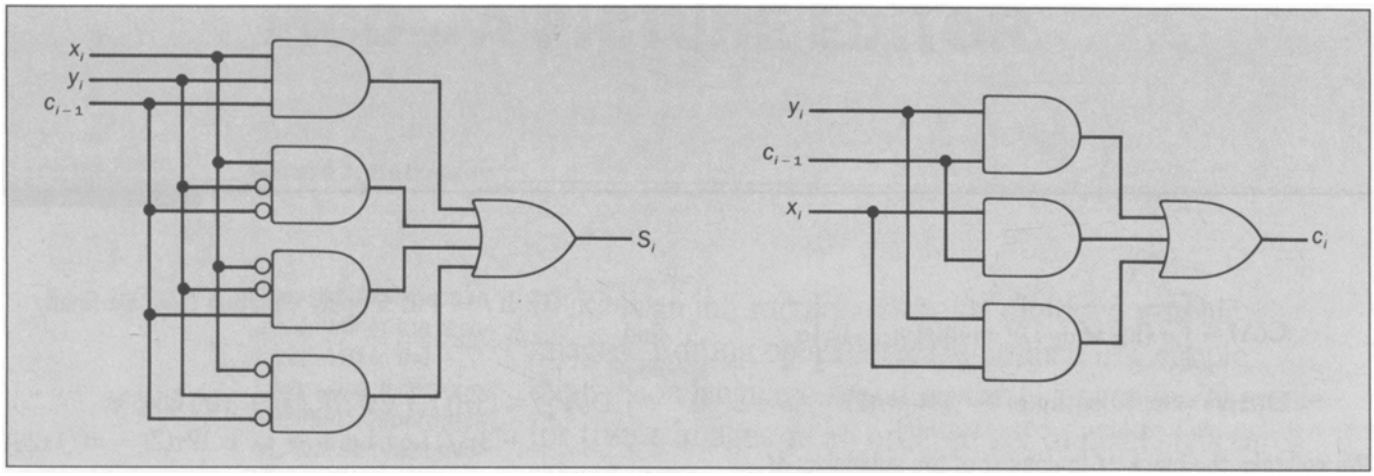

Figure 8. The adder $A_{1,1,2}$.

Lemma 3

If $p(m)$ is any polynomial in m, then there is an m^* that depends on ( ) and d, such that, for $m \geq m^*$, the number of gates required to realize c_m with depth d, even with unlimited fan-in, exceeds $p(m)$.

Chandra, Stockmeyer, and Vishkin[14] have studied reducibility properties of Boolean functions when unlimited fan-in is available. They show that realizing c_m in constant depth is, in a certain sense, as hard as doing the same for multiplication, sorting, the threshold functions, and multiple addition.

The following theorem states our results on the delay of A'_m.

Theorem 4

With fixed fan-in f, the next-state map δ of the automaton A'_m has delay $\mathbf{D}(\delta) > \log_f m + \log_f \lg m$. Even with unlimited fan-in, if δ is represented by m Boolean functions with $\mathrm{Lg}(m+1)$ outputs each, it is impossible to realize δ with both constant delay and complexity polynomial in m.

Proof

Fixed Fan-In. The subtlety here is that we cannot assume anything about how δ is realized, or even about how it is encoded in binary (the state assignment of A'_m). Let $\delta_{a,\phi}$ denote δ_a when the automaton's state is fixed to be ϕ. From an earlier comment, when m is large, this map is $\delta_{a,\phi} : \{0,1\}^m \rightarrow \{0,1\}^{\lceil m \lg m \rceil}$. That is, it consists of at least $m \lg m$ functions. From Theorems 2.4.1.1 and 2.4.4.2[11] over any basis of fan-in f, $\mathbf{C}(\delta_{a,\phi}) \geq (m \lg m - 1) + (m-1)/(f-1))$. The result then follows from Theorem 2.3.3.[11] [$\tau(2)$ or τ does not appear here because Savage assumes that all gates of the basis have delay one.]

Unlimited Fan-In. Here we use the "hardness" results of Lemmas 2 and 3 for c_m. We will show that, if the component $\delta_{a,\phi}$ of δ can be realized with constant delay and polynomial complexity, so can the counting function c_m. However, to reduce the Boolean function c_m to the abstract function $\delta_{a,\phi}$, we need to assume something about how the automaton's states are encoded in binary. (It may be possible to weaken the particular assumption that follows.)

So let $\delta_{a,\phi}$ be the set of functions $\phi_1, \ldots, \phi_m$, with $\phi_k : \{0,1\}^m \rightarrow \{0,1\}^{\mathrm{Lg}(m+1)}$. Then the ϕ_k depend only on $r_1, \ldots, r_m$, just as c_m does. By using one $OR_{\mathrm{Lg}m}$ for each ϕ, we get the *unary* representation of the number of 1s in $r_1, \ldots, r_m$; i.e., a string of c_m 1s followed by 0s. To convert this to the binary representation, we first detect the transition in the string from a 1 to a 0 using $m-1$ XOR_2s, and then use an $(m-1)$-to-Lgm encoder. With unbounded fan-in, this encoder can be built in two levels and with complexity $O(m\mathrm{Lg}m)$. Thus, with unlimited fan-in, c_m can be obtained from $\delta_{a,\phi}$ with depth four greater than that of $\delta_{a,\phi}$, and polynomial complexity.

Complexity of a Level-2 Module

A level-2 module (the automaton B_M) will be implemented along the lines of the previous section. We do not describe the implementation in detail, but use the results of the previous section to estimate the complexity of the realization closely enough to use in the sequel.

The memory of B_M is structured as two queues Q_I

and Q_C, both of length N. Q_I holds identifiers of level-1 modules and is controlled by δ_I, while Q_C holds the corresponding service-duration counts and is controlled by δ_C. The first stage of Q_C is a count-down counter (decrementer) with range 0 to m. From equation (6), the complexity of the two queues is about

$$13\alpha N(\mathrm{Lg}(M+1)+\mathrm{Lg}(m+1))$$

δ_I and δ_C operate in the same way. They can both be realized using a "1s counting" block C_M, a block of N adders $a_{\mathrm{Lg}N+1,\mathrm{Lg}N+1,\mathrm{Lg}N+2,}$, a switch $S_{N,N,\mathrm{Lg}(M+1)+\mathrm{Lg}(m+1)}$, and N decoders $dec(\mathrm{Lg}(N+1))$. From equations (17), (21), (9), and (10), the required complexity is about

$$[3N^2(\mathrm{Lg}(m+1)+\mathrm{Lg}(M+1))+2fN^2 + 28N\mathrm{Lg}N+70M\mathrm{Lg}M]\alpha$$

Using $M = n/m$ to simplify $\mathrm{Lg}(m+1)+\mathrm{Lg}(M+1)$, an estimate of the complexity of B_M is

$$\mathsf{C}(B_M)\approx(3N^2\lg n+2(f+3)N^2+28N\lg N + 13N\lg n+70M\lg M)\alpha \qquad (27)$$

This shows that, if $N = n - 1$, there is *no* advantage to the modular realization because $\mathsf{C}(B_{n/m})$ is $O(n^2\lg n)$, which is the same as $\mathsf{C}(A'_n)$. Clearly, the delay of B_M is

$$\mathsf{D}(B_M) = O(\mathrm{Lg}M+\log_f N) \qquad (28)$$

Modular Design of Large Arbiters

The complexity of an arbiter that is constructed from $M = n/m$ level-1 modules A'_m and one level-2 module B_M is $M\mathsf{C}(A'_m)+\mathsf{C}(B_M)$. Assume that the queues of B_M are limited to length $N = cM$, where c is a constant. If a_n denotes the modular arbiter, then, from equations (25) and (27), its complexity is

$$\mathsf{C}(a_n)\lesssim\frac{n}{m}[3m^2\lg m+(3+2f)m^2]\alpha + c^2\left(\frac{n}{m}\right)^2(3\lg n+2f+6)\alpha \qquad (29)$$

where $\lesssim$ means "asymptotically $\leq$." For a given n, we want to select m to minimize $\mathsf{C}(a_n)$. If we treat m as a real variable and differentiate the right-hand side of equation (29), we find that the minimum occurs when

$$m^3\lg\beta m=\gamma$$

where $\beta=\lg(2+\frac{2f}{3})$

$$\gamma=2c^2n(\lg n+\frac{2f}{3}+2)$$

When $\gamma\to\infty$, the solution to this equation is

$$m^*=\left(\frac{3\gamma}{\lg(3\beta^3\gamma)}\right)^{1/3}\sim(6c^2n)^{1/3} \qquad (30)$$

If we substitute equation (30) into (29), we find that

$$\mathsf{C}(a_n^*)\lesssim[2.73n^{4/3}\lg n + (3.63\lg c+4.24f+12)n^{4/3}]c^{2/3}\alpha \qquad (31)$$

When we compare equation (31) with (25), we see that for large n, the modularization reduces the complexity by a factor of $n^{2/3}$.

This analysis can be generalized to the case $N = cM^{1+\epsilon}$. Then, the dominant term in equation (31) becomes $O(n^{(4+4\epsilon)/(3+2\epsilon)}\log n)$.

Conclusion

We investigated in a top-down manner a first-come, first-served n-process arbiter, neglecting a synchro-

nization problem that, in a strict sense, is known to be intractable. We showed that the circuit complexity of the arbiter was $O(n^2 \log n)$ and its delay was $O(\log n)$, and derived a logarithmic lower bound to the delay. We also exhibited a modular realization of the arbiter, which, under a relaxed equivalence condition, can be implemented with the same delay but smaller complexity.

Two problems related to the modular realization remain. First, we did not show that it is reasonable to assume that the queue length of B_M is proportional to M. Second, where the queue of the level-2 module does overflow, the modular arbiter's behavior will not degrade gracefully. (When the first-come, first-served property cannot be preserved, we would still like to guarantee fairness.) These two problems (the second is perhaps more important) may be areas for further work.

References

1. J. E. Burns, "Symmetry in Systems of Asynchronous Processes," *Proceedings*, 22nd Symposium on Foundations of Computer Science, IEEE, 1981.
2. S. Cohen, D. Lehmann, and A. Pnueli, "Symmetric and Economical Solutions to the Mutual Exclusion Problem in a Distributed System," *10th Colloquium on Automata, Languages, and Programming*, Lecture Notes in Computer Science 154, Springer-Verlag, New York, 1983.
3. A. B. Kovaleski, "High-speed Bus Arbiter for Multiprocessors," *IEEE Proceedings*, Vol. 130, March 1983.
4. G. Färber, "A Decentralized Fair Bus-Arbiter," *Microprocessing and Microprogamming*, Vol. 7, 1981.
5. C. L. Seitz, "Ideas about Arbiters," *LAMBDA* (now *VLSI Design*), 1st Quarter 1980.
6. L. Kleeman and A. Cantoni, "On the Unavoidability of Metastable Behavior in Digital Systems," *IEEE Transactions on Computers*, Vol. C-36, January 1987.
7. Z. Manna and A. Pnueli, "Temporal Verification of Concurrent Programs," *The Correctness Problem in Computer Science*, ed. R. S. Boyer and J. S. Moore, Academic Press, New York, 1981.
8. K. N. Oikonomou, "Ideal Arbiters: Analysis and Design," unpublished manuscript, 1986
9. G. Bochmann, "Hardware Specification with Temporal Logic: An Example," *IEEE Transactions on Computers*, Vol. C-31, March 1982.
10. M. Stucki and J. Cox, "Synchronization Strategies," *Proceedings of the Caltech Conference on VLSI*, January 1979.
11. J. E. Savage, *The Complexity of Computing*, John Wiley & Sons, New York, 1976.
12. G. Broomell and J. R. Heath, "Classification Categories and Historical Development of Circuit Switching Topologies," *ACM Computing Surveys*, Vol. 15, No. 2, June 1983.
13. M. Furst, J. Saxe, and M. Sipser, "Parity, Circuits, and the Polynomial Time Hierarchy," *Proceedings*, 22nd Symposium on Foundations of Computer Science, IEEE, 1981.
14. A. K. Chandra, L. J. Stockmeyer, and U. Vishkin, "A Complexity Theory for Unbounded Fan-in Parallelism," *Proceedings*, 23d Symposium on Foundations of Computer Science, IEEE, 1982.

(Manuscript received February 19, 1987)