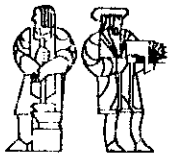
LABORATORY FOR
COMPUTER SCIENCE



MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

The Design of a VLSI Self-timed Ring Buffer Using Signal Transition Graphs

Computation Structures Group Memo 247
May 1986

Tam-Anh Chu

This version supersedes a previous VLSI Memo, No. 85-240 dated March 1985. A short version of this paper entitled "Design of VLSI Asynchronous FIFO Queues for Packet Communication Networks," jointly authored with Clement K.C. Leung, was presented at the International Conference on Parallel Processing 1986.

This research was supported in part by a National Science Foundation Grant 7915255 and a Hughes Fellowship.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

The Design of a VLSI Self-timed Ring Buffer using Signal Transition Graphs [1]

Tam-Anh Chu
Department of EECS, M.I.T.
Cambridge MA 02139

Abstract

Throughput and latency in packet communication networks are determined to a large extent by throughput and latency in the first-in first-out (FIFO) queues used for packet buffering in these networks. We describe a design approach for self-timed FIFO queues with a novel organization which allows tradeoffs between area, throughput and latency in VLSI implementations. This flexibility is made possible by the use of asynchronous distributed control circuits. These circuits are synthesized directly from a graph model called Signal Transition Graphs, and are completely hazard-free. A number of nMOS test chips were fabricated and they worked at a 4 Mbytes/sec throughput rate.

1 Introduction

Multistage interconnection networks (MIN) are used to support communication among processing and storage modules in multiprocessor architectures [2]. To use a packet routing MIN, modules communicate by sending packets to each other. A packet consists of an address and data. The address is used to forward the packet along a path through the network from the sender to the destination module. Multistage interconnection networks are usually constructed out of $N \times N$ crossbar switches, where the number of ports N is determined by performance requirements and packaging constraints. A switch forwards an input packet to an output port according to the destination address of the packet. Each switch may also provide buffering storage for packets whose forwarding paths are temporarily blocked by other network traffic. The buffering storage is usually managed by a first-in first-out discipline.

In this setting, the design of FIFO queues is an important consideration in the design of practical packet routing networks. We present an approach for designing FIFO queues in VLSI technology which allows tradeoffs between area, latency and throughput. At one end of the design spectrum, an area-efficient implementation with high throughput, but long latency, can be obtained. In this organization, the register stages are connected serially to allow data to ripple through; there is no global communication. At the other end of the spectrum is a queue with minimal latency, but somewhat lower throughput rate due to

[1] This version supersedes a previous CSG Memo. No. 247 dated March 1985. A short version of this paper entitled "Design of VLSI Asynchronous FIFO Queues for Packet Communication Networks," jointly authored with Clement K.C. Leung, is presented at the International Conference on Parallel Processing 1986.

increased delay in the control operations and in the loading of global buses. On a VLSI chip, these buses are sets of wires carrying input and output data which are connected to the input and output of all register stages. An optimal design somewhere in the range of these extremes can be chosen depending on the application. A queue of N stages can be partitioned such that only M register stages load the buses; given that the permissible latency is L stage delays, then $M = N/L$. Thus, this distributed organization also reduces the amount of global communication; this is particularly important for large queues, where the loading on control and data buses approaches the level existing in a typical memory array.

Our FIFO queue design makes use of distributed control structures and local communication. There are only a few types of modules in this design, with modules of each type replicated as necessary to construct complete FIFO queues. The distributed control structure allows the exploitation of concurrency. Concurrent read/write supports a higher throughput rate. The FIFO queue is also completely data driven, hence no potential read/write conflict exists and there is no need for any arbiter.

The distributed control organization of the FIFO lends itself naturally to a design using asynchronous, self-timed hardware circuits. Towards this end, a specification technique for asynchronous control structures based on a graph model called Signal Transition Graphs (STGs) have been proposed, and methods for direct and efficient synthesis of self-timed hardware circuits from such specifications have been developed. The STG model is especially appropriate for control modules which exhibit a high degree of asynchronous concurrency. A preliminary discussion of the STG model and its expressiveness and implementability are given in [4]. The use of STGs in the design and implementation of a self-timed 2×2 packet router is reported in [3].

The paper is organized as follows. Section 2 describes the functional behavior and alternate organizations of the FIFO queue. Section 3 introduces the STG model and discusses its use in the specification of self-timed circuits. In Section 4, STG specifications of the building block control modules for realizing the various FIFO queue organizations and their implementation are presented. Finally, Section 5 presents the results and some further discussions on the STG model.

2 Organization of the FIFO Queue

This section discusses the one and two-dimensional organizations for the FIFO queue (Figs. 1 and 2). The R-module is a control circuit designed to support pipelined operation of the register stages. A R-module has an input link from a previous stage, an output link to the next stage, and an output link to a register module in the same stage to control data loading into this register. A *link* is a pair of *ready/acknowledge* wires, depicted as an arc with the arrow pointing in the direction of the *ready* signal. When input data is available for loading into a register, a *ready* signal is sent to its R-module controller on the

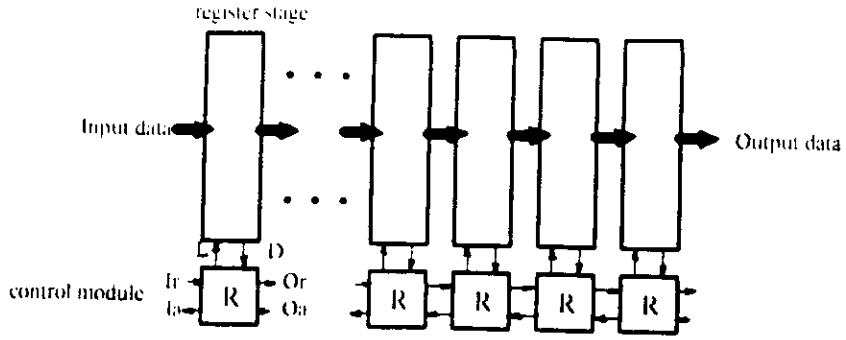


Figure 1: Organization of a one-dimensional (linear) FIFO queue.

I_r wire (Fig. 1). The actual loading is performed when the R-module controller sends a *ready* signal to the register on the L wire. Once data have been loaded into a register, an *acknowledge* signal is sent to its R-module on the D wire. The R-module then returns an *acknowledge* signal on the I_a wire of the input link and forwards a ready signal on the O_r wire of its output link concurrently. The next data item will be loaded into its register only after the R-module has received an *acknowledge* signal on its O_a wire and another *ready* signal on its I_r wire. Thus, the operation of the R-module is pipelined. Data from one stage will be forwarded to the next unless the latter is full. The throughput rate of this queue is determined by the delays of the R-module and registers, whereas its latency is proportional to the number of stages in the queue.

A two-dimensional, or *ring* organization is shown in Fig. 2. This queue consists of M linear queues, each of L stages, and two *token rings* for controlling input/output operation. The capacity of the queue is $M \times L$ and the latency is proportional to L. I-modules are connected together to form a *token ring* to control the writing of data into the queue. The ring is initialized such that only one I-module contains the token, marking the next available empty register stage. Since the *Write-request* signal, carried on wire W_r, is connected to all I-modules, the token should not be passed on to the next module in the ring if the *Write-request* signal is still active. This is an important timing restriction. The *Write-acknowledge* on wire W_a is the output of an OR gate (shown as a heavy bar with a + sign) whose inputs are acknowledge wires from all I-modules. Similarly, reading from the FIFO is controlled by an *Output token ring*, formed by connecting O-modules together. Data written into the linear queues ripple to their output side, ready to be gated onto the output bus. The Output ring is initialized such that only one O-module contains the token. This module then controls the timing and signaling for gating of data to the output bus; the detailed organization is explained in Section 4.2. The *Read-request* signal on wire R_r is the output of an OR gate whose inputs are request wires from all O-modules. Another timing restriction exists for the Output token ring: since the *Read-acknowledge*

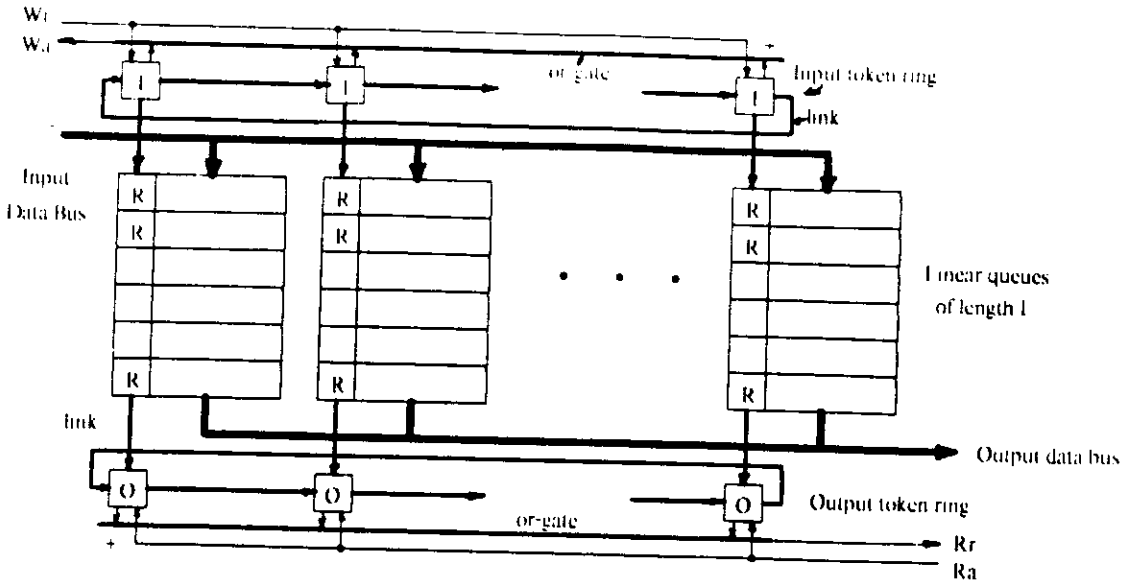


Figure 2: The ring organization of a FIFO queue

signal, carried on wire R_a is broadcast to all O-modules, the token should not be passed on to the next module while *Read-acknowledge* is still active.

The Ring buffer which we fabricated is one with minimal latency ($L = 1$), with each of the linear queues containing exactly one stage. Registers in each stage have inputs connected to the input data bus, and outputs connected to the output data bus.

3 Signal Transition Graphs

In this section we introduce the STG model and its application to the specification of pipeline controllers. A more complete discussion of STG can be found in [5]. In short, STGs are a form of Petri nets restricted by a set of axioms, and their components (such as transitions and places) assigned attributes related to physical circuits. The result is a graph model which is much more amenable to analysis due to the reduced complexity, and still retains sufficient expressiveness for specifying most common behaviors of control circuits including concurrency, choices and conflicts.

3.1 The STG notation

A hardware circuit consists of an interconnection of *logic elements*, each having an output terminal and a number of input terminals. Every input terminal is connected either to an input terminal of the entire circuit, or to an output terminal of another logic element in the circuit. The set of all terminals of a circuit is call the set of *signals* M. In order to describe the dynamics of a circuit, a set of *signal transitions* $T = M \times \{+, -\}$ is used to specify the rising and falling transitions of each signal in M. For each $i \in M$, its associated

transitions are denoted by $i+$ and $i-$. It is often convenient to use the notations t and $\bar{t}$ to denote pairs of transitions, such that if $t = i+$ then $\bar{t} = i-$ and vice versa.

For the purpose of this presentation, a STG is a directed graph represented as a triple $\langle T, T_{E0}, \mathcal{R} \rangle$ where T is the set of signal transitions (defined over a signal set M), T_{E0} the set of transitions which are enabled in the initial state of the circuit, and $\mathcal{R} \subseteq T \times T$ an *irreflexive, intransitive* relation over the set of transitions, called the *causal* relation. Graphically, $t_1 \mathcal{R} t_2$ is shown as an arc between two transitions: $t_1 \rightarrow t_2$. Let $\mathcal{R}^+ \subseteq T \times T$ denote the transitive closure of $\mathcal{R}$, $t_1 \mathcal{R}^+ t_2$ means that there exists a directed path from t_1 to t_2; this is shown graphically as $t_1 \rightarrow_p t_2$. The semantics of STG can be expressed in terms of *transition sequences* and their compositions; this has been carried out in [5] using trace theory [13]. Informally, $t_1 \mathcal{R} t_2$ means that the occurrence of transition t_1 causes that of transition t_2; this implies that if the circuit is in some state s in which transition t_1 is enabled and eventually occurs, then the occurrence of t_1 brings the circuit to another state s' say, in which t_2 is enabled and hence will eventually occur. This last statement hints that given an initial state of the circuit (in which transitions in T_{E0} are enabled) and a STG expressing the causal relation between its signal transitions, one can generate a state transition graph from the STG. A circuit realization can then be obtained from the state graph. Furthermore, a constraint $t_1 \mathcal{R} t_2$ can be implemented as a logic element with t_1 as one of the inputs and t_2 as output. This important observation allows the decomposition of a STG specification into smaller subgraphs, each of which contains only transitions which are causally related. Thus, STG allows a very efficient and direct implementation based on this decomposition principle. This is a unique feature of STG compared to other approaches.

A multi-arc connects several tail transitions to one head transition, or one tail transition to several head transitions. We call these arc configurations *And Forks* and *Joins* (there are also *Or* constructs for specifying choices or conflicts in the complete STG model), and their diagrammatic notations are shown in Figure 3a. An And-fork is used to describe a situation in which the occurrence of a tail transition causes the occurrences of all of the head transitions. An And-join describes a situation in which all tail transitions in the relation have to occur to cause the occurrence of the head transition. These And constructs are used to describe concurrent operations in circuits.

3.2 Liveness and Persistency

A STG has a deadlock-free and hazard-free circuit realization only if it satisfies the properties of liveness and persistency. Since STGs are merely behavior specifications from which state graphs can be derived for implementation, these properties must ultimately be based on the latter type of graphs. However, there is a one-to-one correspondence between them, so that liveness and persistency will appear as *syntactic* constraints on STGs. We discuss briefly these properties and their STG syntactic ramifications .

The continual operation without deadlocking of control modules is a property called

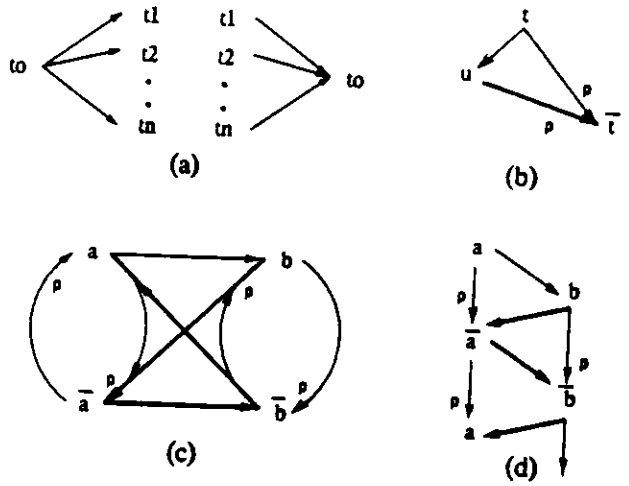


Figure 3: (a) Conjunctive Forks and Joins. (b) A persistency constraint. (c) A specification of pipelined operation. (d) The partial order resulted from "unfolding" of the STG in (c).

liveness. A STG is live iff its underlying state transition graph is strongly connected. The *necessary* condition for a STG to be live consists of (i) the STG is a strongly connected graph, and (ii) there is a *simple* cycle containing both t and $\bar{t}$ for every $t \in T$. Since live STGs are strongly connected, concurrency and ordering have to be characterized differently: two transitions can occur concurrently iff there is no simple cycle containing both of them in the STG; equivalently, the occurrence of two transitions are ordered iff there exists a simple cycle containing both of them.

Due to the similarity to a special class of Petri nets called *marked graphs* [6], it may appear that the form of STGs discussed here is always persistent. However this is not the case, as the underlying state graph may exhibit nonpersistency whenever two transitions are enabled in the same state and the occurrence of one removes the enabling condition of the other. A *persistency constraint* is an ordering constraint between two transitions used to eliminate nonpersistency, as illustrated in Fig. 3b. For a *live* STG, the condition $t\mathcal{R}^+\bar{t}$ always holds for every transition t. If $t\mathcal{R}u$ exists as shown and $u\mathcal{R}^+\bar{t}$ (depicted as an heavy arc in Fig. 3b) were not present, then transitions $\bar{t}$ and u can occur concurrently. Suppose the course of action $t\mathcal{R}u$ is implemented by a hardware element with t as one of its inputs and u as its output. Concurrency between $\bar{t}$ and u implies that while the hardware element is reacting to t to cause u, $\bar{t}$ may be occurring simultaneously at the input of that hardware element. This is commonly known as a race condition in hardware circuits and can lead to malfunction. The approach to deal with this problem is to impose a persistency constraint on STG specifications, namely $u\mathcal{R}^+\bar{t}$, to eliminate this nonpersistence behavior. Hence, a STG specification is persistent if every transition u caused by a transition t precede $\bar{t}$, i.e.

$\forall u \in T$, if $t\mathcal{R}u$ then $u\mathcal{R}^{+}\bar{t}$.

3.3 Specification of Pipelined Circuits.

We can now develop a STG specification for pipelined control operations such that liveness and persistency are satisfied. Consider two cycles of transitions as shown in Fig. 3c. The left cycle (with a and $\bar{a}$) represents the control sequence of the input portion of a pipelined circuit; the right one (with b and $\bar{b}$) represents the control sequence of its output portion. The necessary condition for liveness is satisfied by each cycle if every transition in a cycle is paired with another in the same cycle. We want the two cycles to operate in parallel as much as possible, with the left one initiates control actions on the right one through arc $a\mathcal{R}b$. In order for the STG to be persistent, three additional arcs (in heavy line) are required. Because of the existence of arc $a\mathcal{R}b$, arc $b\mathcal{R}\bar{a}$ is required as a persistency constraint to prevent concurrent firing of b and $\bar{a}$. The introduction of $b\mathcal{R}\bar{a}$ requires adding $\bar{a}\mathcal{R}\bar{b}$ to prevent concurrent firing of $\bar{a}$ and $\bar{b}$, and this in turns requires arc $\bar{b}\mathcal{R}a$. These four arcs allows the synchronization of two cycles of transitions in pipelined fashion such that the resulting STG is persistent. These constraints can be viewed from a different perspective by "unfolding" the cycles (Fig. 3d) into partial orders, in much the same fashion as occurrence nets [1]. In this type of nets, a node such as a represents an *instance* of a transition a in a cycle of operation. It can be seen that the persistency constraints appear as $a\mathcal{R}b\mathcal{R}\bar{a}\mathcal{R}\bar{b}\ldots$ and thus forces these transitions to occur in sequence. Otherwise, transitions belonging to other branches of the cycles can occur concurrently.

4 STG specification of control modules

In this section we apply the STG model to specify the control modules used in the FIFO organizations in Section 2. We will illustrate how to specify a STG such that it meets the liveness and persistency conditions set forth in Section 3. Once a STG satisfying these conditions is obtained, it can be translated directly into circuit modules, using the synthesis steps discussed in [5]. We will give the hardware implementation obtained through this synthesis procedure for each STG specification, and discuss the procedure itself informally.

For all the control modules we will specify, event occurrences are signalled over control links, using the *reset signaling* handshake protocol [11]. Usually, an occurrence of an event is signalled by a positive transition on the ready wire of the control link; its acknowledgment is signalled by a positive transition on the acknowledge wire of the control link. The signals on these links are then reset through negative transitions before the occurrence of the next event can be signalled.

While liveness and persistency are considered to be fundamental properties of STG, there are other properties more related to the implementation of control circuits according to a certain design methodology. Two such constraints pertinent to the ensuing discussions called **R1** and **R2** are described.

R1 This constraint concerns the behavior in the initial state of a control circuit operating with the reset signaling protocol. Starting from the idle initial state, every control module used in the FIFO organizations alternates between an active phase consisting entirely of positive signal transitions, and a reset phase consisting entirely of negative signal transitions. In a circuit implementation, the signal state at each terminal is identified with a signal transition at that terminal: if the state of a signal u is 1 (0), it implies that $u+$ ($u-$) has occurred. If the initial state of a circuit is all 0's then negative transition of the form $u-$ must have occurred in each of these signals in the immediate past. Thus, any positive transition in a STG, say $u+$ which is preceded only by negative transitions of the form $t-$ will always be activated in this initial state. When this is not desired, an artificial constraint from some other positive transition $r+$ to $u+$ must be added. Hence, it is required that *for every STG, the subgraph induced by the set of positive transitions is connected.*

R2 The second constraint results from the communication discipline imposed on a control circuit. Control circuits operating under the reset signaling protocol uses pairs of *ready/acknowledge* wires to communicate with the external world. A transition on the *acknowledge* wire can only occur in response to a transition on the *ready* wire and vice versa. For a pair of wires $\{I_r, I_a\}$ where I_r is an input *ready* and I_a an output *acknowledge*, this communication interface to the external world is specified in a STG by the pair of constraints $\{I_a - \ \mathcal{R}I_r+, I_a + \ \mathcal{R}I_r-\}$. Similarly for a pair of wires $\{O_r, O_a\}$ where O_r is an output *ready* and O_a an input *acknowledge*, its corresponding set of constraints is $\{O_r+ \ \mathcal{R}O_a+, O_r- \ \mathcal{R}O_a-\}$. Thus, in a STG, *every transition of an input signal has exactly one transition which directly precedes it, and this transition must be that of an output signal.* Transitions of an input signals are underlined to distinguish it from transitions of "non-input" ones.

The remaining of this section describes the operation of the I-,O- and R-modules and their STG specifications. It will be seen that their STGs satisfy the liveness and persistency properties as discussed previously.

4.1 The I-module

An I-module controls the loading of input data into a linear queue (Fig. 2). When its turn comes, a token is passed to the I-module from its immediate predecessor in the token ring, through a control link $P = \{P_r, P_a\}$. Upon receiving the token, the I-module responds to the next input request on its $W = \{W_r, W_a\}$ control link by sending a load request to the linear queue it controls through its $I = \{I_r, I_a\}$ control link. After loading a new data item into the linear queue, the I-module forwards the token it holds to the next I-module in the token ring through its $N = \{N_r, N_a\}$ link.

The STG description of I-module in Fig. 4 contains two main cycles, the left one coordinates the reception of the ring token with the reception of the next data item presented to the FIFO queue. The right one manages the forwarding of ring token to a successor

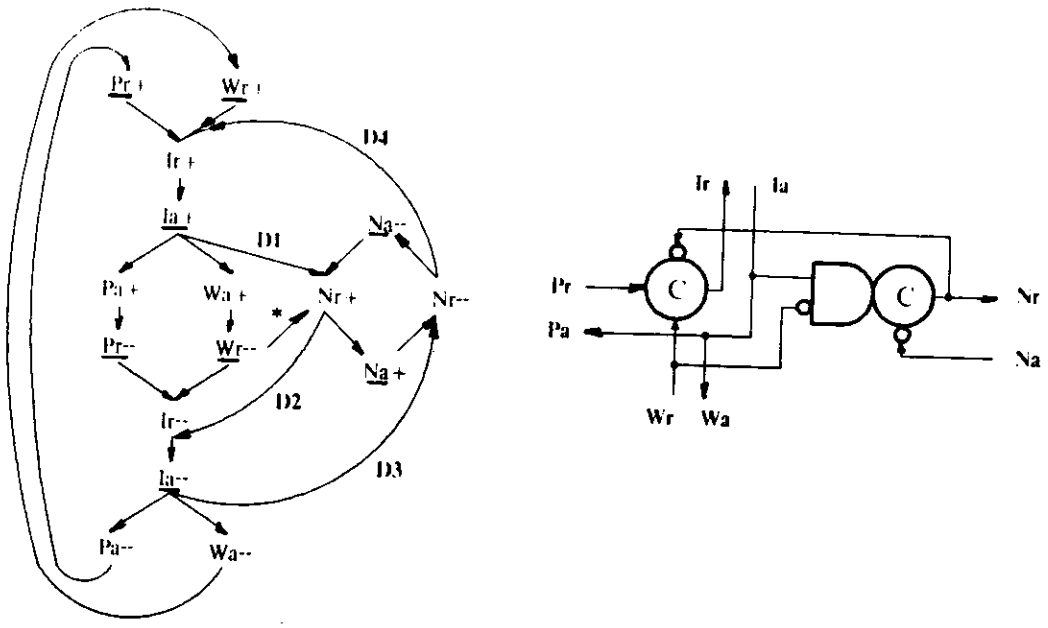


Figure 4: STG description and realization of the I-module

I-module. The *-arc ($W_r - \mathcal{R}N_r+$) implements the timing constraint discussed earlier, such that N_r does not go high (to pass a token to the next module) until after the input *Write-request* W_r has gone low. These two cycles and the *-arc together provide the specification for proper event sequencing in the I-module. Other arcs are to be added to satisfied persistency and other constraints discussed above. First, arc D_1 ensures that all positive transitions form a connected subgraph according to constraint **R1**. Since D_1 is a constraint from transition I_a+ to N_r+, the pairs of transitions $\{I_a+, I_a-\}$ and $\{N_r+, N_r-\}$ could be used for implementing the persistency constraints. This set of arcs would include: $I_a + \mathcal{R}N_r+$, $N_r + \mathcal{R}I_a-$, $I_a - \mathcal{R}N_r-$, and $N_r - \mathcal{R}I_a+$. However, since I_a+ and I_a- are transitions of an input signal, each can have no more than one incident arc according to constraint **R2**. To enforce these constraints, we change $N_r + \mathcal{R}I_a-$ to $N_r + \mathcal{R}I_r-$, and, $N_r - \mathcal{R}I_a+$ to $N_r - \mathcal{R}I_r+$. These final constraints are shown as $D_1 - D_4$ in Fig. 4.

Liveness and persistency are satisfied by this STG. The synthesis procedure produces a state graph, from which the realization in Fig. 4 is obtained. The logic equation for I_r is $I_r = W_r P_r \overline{N}_r + I_r(W_r + P_r + \overline{N}_r)$ and its implementation is a C-element with inputs W_r, P_r and $\overline{N}_r$. The logic equation for N_r is $N_r = \overline{N}_a \overline{W}_r I_a + N_r(\overline{N}_a + I_a)$ and its implementation is as shown in Fig. 4. The reader can readily verify that the circuit operates according to its STG specification.

4.2 O-module

The O-module controls the gating of output data in one of the linear queues onto the output bus (Fig. 5). Its operation is somewhat similar to that of the I-module. When

its turn comes, a token is passed to an O-module from its immediate predecessor in the token ring, through a control link P. The O-module which receives the token will respond to an *output ready* on link O from its linear queue by sending a *gating request* on link G (this link is not shown in Fig. 2). After data have been gated onto the output bus, the O-module sends a *Read request* to the external world on link R. Upon acknowledgment from the external world, it forwards the token it holds to the next O-module in the token ring though its N link.

As in the STG for the I-module, the STG for the O-module also has two main cycles. The one on the right forwards the ring token. The one on the left coordinates the receipt of the ring token and the receipt of a new piece of data from the linear queue with the gating of this new piece of data onto the output bus and the reading of this new piece of data from the output bus by the outside world. During the reset phase, G_r- (Fig. 5a) disconnects the output of the linear queue from the output bus, the R control link is then completely reset before the token is passed on to the next O-module on the token ring via N_r+. This latter sequencing is necessary because R_a is fed to all O-modules. The constraint $R_a + \mathcal{R}N_r+$ is added so that all the positive transitions form a connected subgraph to satisfy constraint **R1**.

The STG in Fig. 5a has an inconsistency in that R_a+ and R_a- both precede N_r+, which requires the signal at node R_a in the hardware implementation to be both at 0 and at 1 in order for signal N_r to undergo a positive transition. We eliminate this inconsistency by introducing nodes $y+$, and $y-$ as in Fig. 5b. Additional arcs emanating and terminating at node y are also introduced to satisfy the persistency constraints for proper pipelining between the two cycles.

If the STG in Fig. 5a were implemented directly according to our techniques, the hardware circuit whose output is P_a will have as inputs R_a and P_r. In the implementation obtained from Fig. 5b, this hardware circuit will have only a single input y. Introducing nodes can lead to simplified implementations much as eliminating common subexpressions in computer programs. In Fig. 5c, another pair of nodes $x+$ and $x-$ are introduced, reducing the inputs to the circuit for generating y from 4 to 3, and reducing the inputs to the circuit for generating G_r from 3 to 2. Implementation of the STG in Fig. 5c is shown in Fig. 6.

4.3 R-module

The R-module is a pipelined module which controls the loading of data into a register. As shown in Fig. 7, it has an input link I, an output link O, and communicates with a register via the control signals L and D. The arrival of input data is signalled by I_r+. $L+$ initiates the loading of this new data into the register. The completion of data loading is indicated by $D+$. Signal D is not an input signal to the R-module in this design, rather it is generated by some internal delay mechanism to match the delay of data latches used in the registers. This delay mechanism is further detailed below.

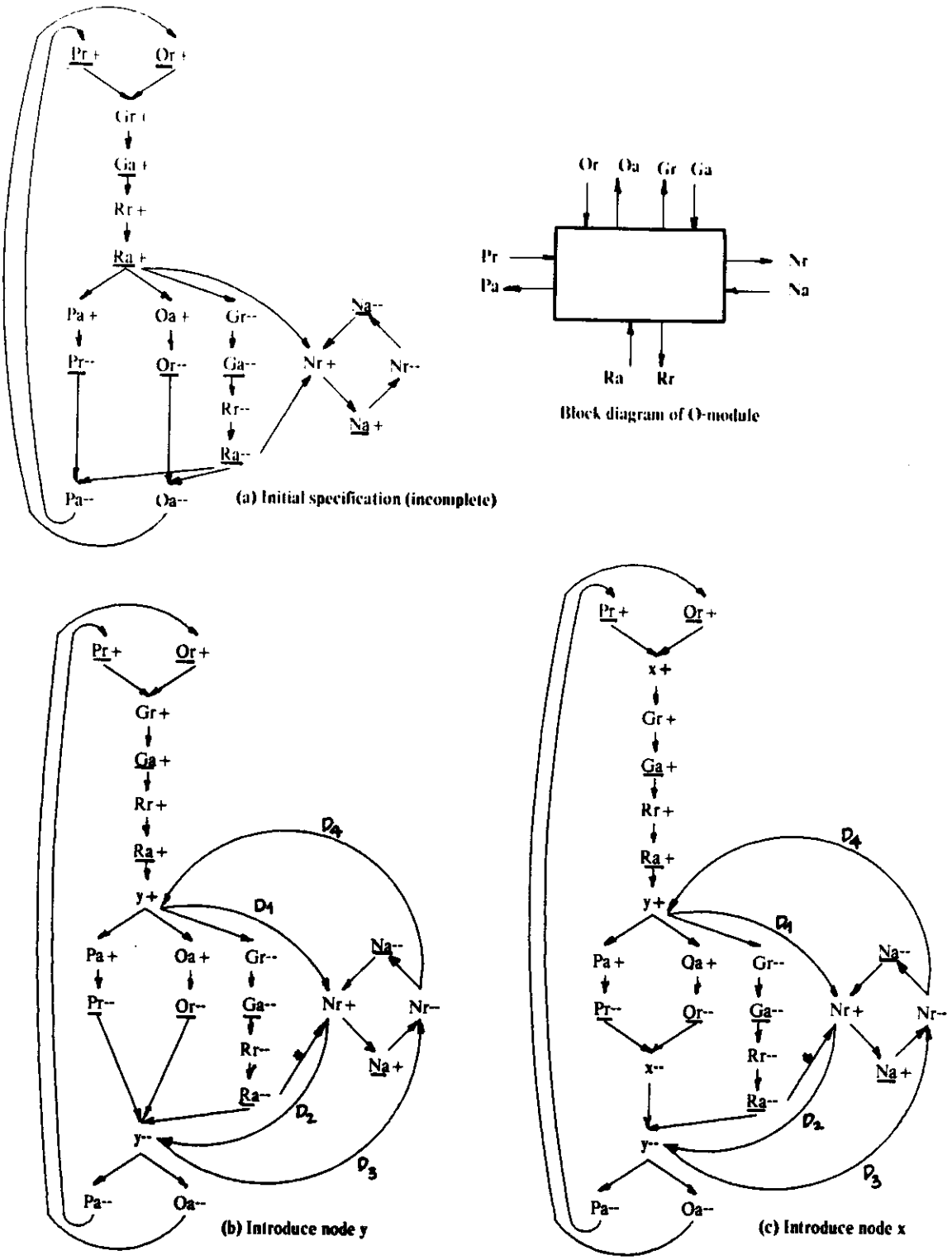


Figure 5: Refinements of STG specifications of O-module.

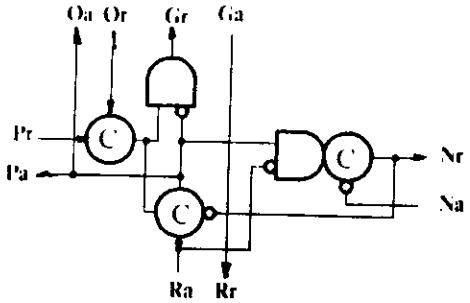


Figure 6: Implementation of O-module.

The STG specification of the R-module (Fig. 7a) contains two cycles: the left one contains transitions of link I and signals L and D, the right one contains those of the output link O. After $D+$ occurs, indicating that data have been stored, the R-module turns off the load signal L before raising the acknowledge signal I_a; this allows the input data to change only after the registers have disconnected their input gates. The *-arc implements constraint **R1**. Transition $D+$ also causes the R-module to raise output ready O_r through arc D_1. Persistency could then be satisfied with the set of arcs $D+ \; \mathcal{R}O_r+$, $O_r+ \; \mathcal{R}D-$, $D_r- \; \mathcal{R}O_r-$ and $O_r- \; \mathcal{R}D+$. There is one other sequencing constraint that needs to be satisfied, however. After O_r+ occurs, transitions $D-$ and I_a- take place in succession, initiating another input cycle. If the input side operates "faster" than the output side, then data overwrite can be prevented by inhibiting the next $L+$ until after the current output has been acknowledged through O_a+. Note that this constraint $O_a+ \; \mathcal{R}L+$, when added to the STG, also enforces the last constraint $O_r- \; \mathcal{R}D+$ needed to ensure persistency, which can now be removed.

From this graph, the synthesis procedures are applied to obtain a circuit implementation as shown in Fig. 7b. The logic equation for D is $D = L + D(R_i + \overline{R}_o)$ and its implementation is a RS flipflop with the reset input being $R_o.\overline{R}_i$. This circuit suggests that the delay of the RS flipflop can be used to "time" the loading operation of data registers. This technique works well in integrated circuits because delays of similar components track each other in a wide range of voltage and temperature variation. In an nMOS implementation, as shown in Fig. 7c, the RS flipflop is implemented as a standard pseudo static register. This register can be conveniently reset by pulling node D low directly through a *pull-down* transistor (the small triangle is the symbol for Ground). When L goes high, it turns on the *pass* transistor which connects a high voltage to the internal input node of the register. After some delay, node D is set to high. Signal $H = \overline{L}$ is used to close the feedback loops when L is low. The data registers have the same configuration and are controlled by signals L and H. The timing diagram for the R-module is shown in Fig 7d.

5 Result and Conclusion

A FIFO with 8 stages and 9-bit wide data path was designed, using a 4 micron nMOS technology. The chip size (including pads) is $3.15 \times 2.25mm^2$. Six chips were received from MOSIS, they were tested and five were fully operational at a throughput rate of approximately 4 MBytes/sec. An nMOS circuit diagram and a photomicrograph of the chip are shown in Fig. 8, the lower portion is the control circuitry, with R-modules on top, I-ring in the middle and O-ring at the bottom. The control circuits take a relatively large amount of area in this chip. However, in a 2-dimensional organization with $L \geq 2$, the overhead due to I-modules and O-modules can be reduced significantly.

In this paper, Signal Transition Graphs have been used as a specification tool for asynchronous control modules. A STG specification can be viewed as an interpreted Petri net in which each transition is identified with a signal transition in a hardware circuit. In the synthesis approach proposed, a state transition graph is generated from a STG and then is used to derive logic equations and hardware structures for the signals. A STG specification can thus also be viewed as a concise yet more abstract notation for specifying a class of state transition graphs.

In our specification and design examples, it has been shown how introducing additional constraints in a STG allows us to use level-sensitive hardware circuits instead of transition-sensitive hardware circuits in its implementation. These constraints are justified only informally. A more formal theory based on trace theory and state transition graphs are developed in [5].

The module descriptions used in this paper require only constructs for specifying sequencing and concurrency. There are other behaviors which exhibit conflict and data-dependent signal flow that would require additional STG constructs for their specification. These latter constructs are called OR-constructs, and the reader is referred to [4] for an introduction to their formulation and applications.

In [8] Martin described a design approach using constructs for non-deterministic programming to specify hardware modules whose behaviors exhibit only sequencing and arbitration requirements. This approach uses a subset of Dijkstra's guarded command language to specify each process; concurrent cooperating processes are described using notations similar to Hoare's CSP [9]. Heuristic procedures are used to "compile" a hardware implementation from a module specification into an interconnection of standard hardware templates such as And, Or, C-elements, etc.. During the compilation, the technique of reordering signal transitions in a sequence is made use to improve implementation efficiency. The complete STG model allows the specification of concurrency, sequencing and conflict in module behavior, and our implementation approach is aimed at automating the derivation of hardware structures from STG specifications. Recently there are works on the classification and synthesis of delay-insensitive circuits based on trace theory [12,13]. The relation of STG to trace theory is analogous to that of Petri nets model to its underlying sequence semantics. Thus, we believe that STG can serve as a high-level, more

abstract specification than that of an approach directly based on trace theory.

There are also related works on verification of asynchronous hardware structures based on temporal logic [7]. Such techniques can be used fruitfully for correctness validation of self-timed circuits. The design of suitable translation techniques from high-level language to STG – in the same vein as those done by Martin and Rem [10] – is another area for further research.

Acknowledgments

The author would like to thank Prof. Jack Dennis for his support and guidance, Prof. Lance Glasser and Dr. Bill Ackerman for helpful suggestions. This research was supported in part by NSF-7915255 and a Hughes Fellowship.

REFERENCES

1. Best, E "Concurrent Behaviour: Sequences, Processes and Axioms." *LNCS 197*, Springer-Verlag 1984.

2. Chin, C.-Y. and K. Hwang, "Packet Switching Networks for Multiprocessors and Data Flow Computers." *IEEE Transactions on Computers*, C-33, No. 11, November 1984.

3. Chu, T-A., C. Leung and T. Wanuga, "A design methodology for concurrent VLSI systems." *Proceedings of the ICCD-85*, Oct. 1985.

4. Chu, T-A. "On the models for designing VLSI asynchronous digital systems." To be published in *INTEGRATION, the VLSI Journal.* North-Holland, 1986.

5. Chu, T-A. *Signal Transition Graphs and the Modeling of Self-timed Circuits.* Ph.D. thesis, Department of EECS, MIT, expected Sept. 1986.

6. Commoner, et. al. "Marked directed graphs." *Journal of Computer and System Sciences.* No. 5, 1971.

7. Dill, L.D. and E.M. Clarke, "Automatic verification of asynchronous circuits using temporal logic." *Proceedings of the 1985 Chapel Hill Conference on VLSI.* Computer Science Press, May 1985.

8. Martin, A.J. "The design of a self-timed circuit for distributed mutual exclusion." *Proceedings of the 1985 Chapell Hill Conference on VLSI.* Computer Science Press, May 1985.

9. Martin, A.J. "Compiling Communicating Processes into Delay-Insensitive VLSI Circuits." 5210:TR:86, Dept. of Computer Science, CalTech 1986.

10. Rem, M. "Concurrent Computations and VLSI Circuits." in *Formal Description of Programming Concepts - II.* D. Bjørner (ed.), North-Holland Pub. Co. IFIP 1983.

11. Seitz, C.L. "System Timing." Chapter 7 of *Introduction to VLSI Systems*, Mead and Conway (Eds.), Addison Wesley 1981.

12. Udding, J.T. *Classification and Composition of Delay-Insensitive Circuits.* Ph.D. thesis, Dept. of Mathematics and Computing Science, Eindhoven Univ. of Technology, 1984.

13. van de Snepscheut, J.L.A. *Trace Theory and VLSI Design. LNCS 200*, Springer-Verlag 1985.

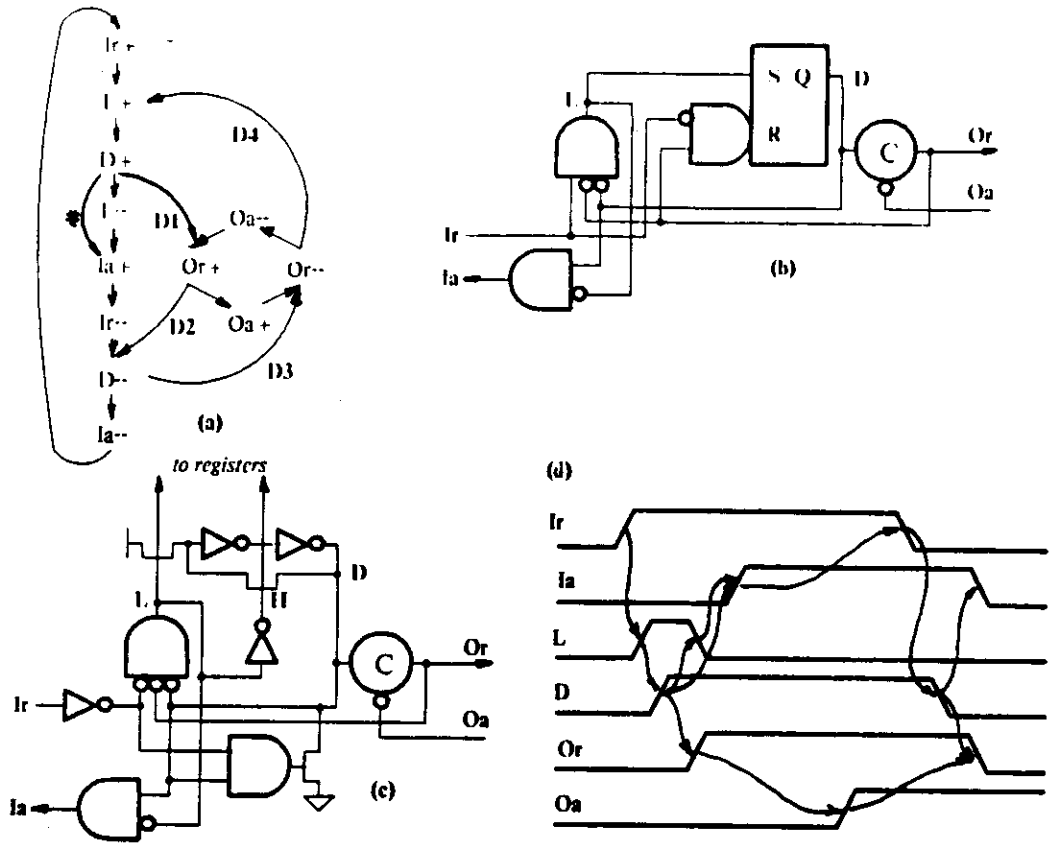
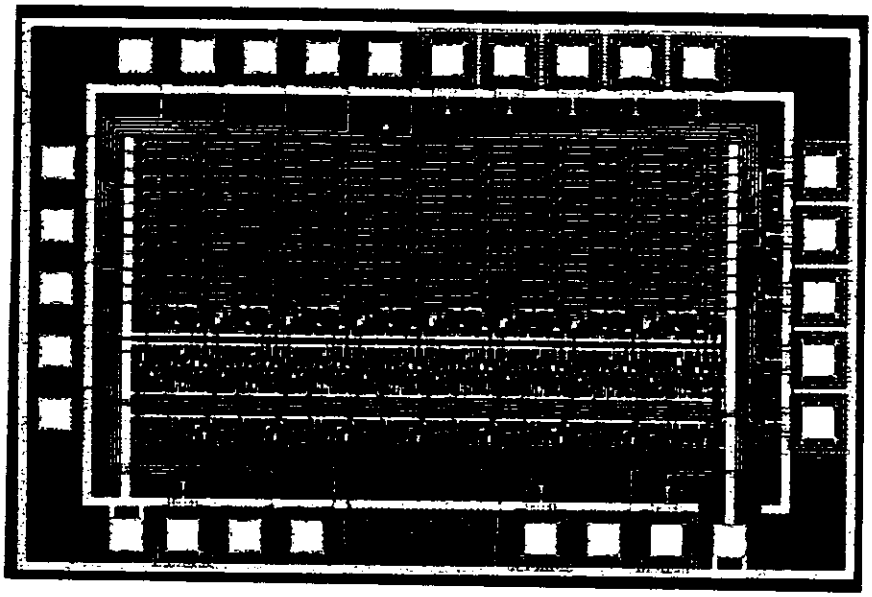


Figure 7: (a) STG specification of R-module, (b) its implementation, (c) an nMOS implementation interfacing to registers, (d) its timing diagram.



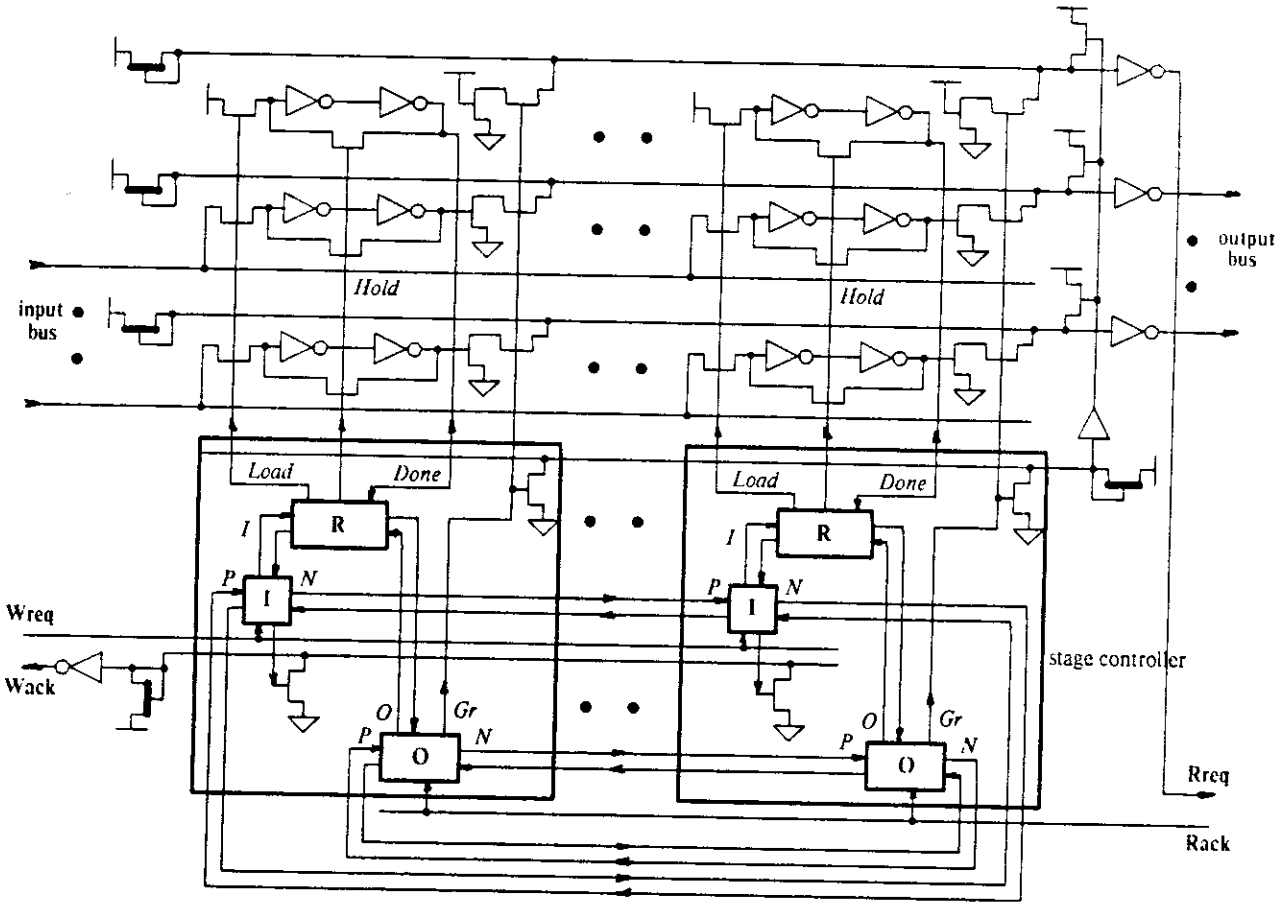


Figure 8: Photomicrograph and circuit diagram of the test Ring Buffer.