CMU-CS-85-102

Compiling Path Expressions into VLSI Circuits

T. S. Anantharaman
E. M. Clarke
M. J. Foster†
B. Mishra

Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

August 1984

†Current address: Department of Computer Science, Columbia University, New York, New York 10027.

This research was partially supported by NSF Grant MCS-82-16706, and the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-81-K-1539.

Table of Contents

Abstract: Path expressions were originally proposed by Campbell and Habermann [1] as a mechanism for process synchronization at the monitor level in software. Not unexpectedly, they also provide a useful notation for specifying the behavior of asynchronous circuits. Motivated by this potential application we investigate how to directly translate path expressions into hardware.

Our implementation is complicated in the case of multiple path expressions by the need for synchronization on event names that are common to more than one path. Moreover, since events are inherently asynchronous in our model, all of our circuits must be self-timed.

Nevertheless, the circuits produced by our construction have area proportional to $N \cdot \log(N)$ where N is the total length of the multiple path expression under consideration. This bound holds regardless of the number of individual paths or the degree of synchronization between paths.

1. Introduction

As the boundary between software and hardware grows less and less distinct, it becomes increasingly important to investigate methods of directly implementing various programming language features in hardware. Since many of the problems in interfacing hardware devices involve some form of process synchronization, language features for synchronization deserve considerable attention in such investigations. In this paper we consider the problem of directly implementing path expressions as self-timed VLSI circuits. Path expressions were originally proposed by Campbell and Habermann [1] for restricting access by other processes to the procedures of a monitor. For example, the simple readers and writers problem with two reader processes and a single writer process is solved by the following multiple path expression:

path R_1 + W **end,**
path R_2 + W **end.**

The first path expression prohibits a read operation by the first process from occurring at the same time as a write operation. The second path expression enforces a similar restriction on the behavior of the second reader process. In a computation under control of the multiple path expression, the two read operations may occur simultaneously, but a read and write operation cannot occur at the same time.

Path expressions are useful for process synchronization for two reasons: First, the close relationship between path expressions and regular expressions simplifies the task of writing and reasoning about programs which use this synchronization mechanism. Secondly, the synchronization in many concurrent programs is finite state and thus, can be adequately described by regular expressions. For precisely the same reasons, path expressions are useful for controlling the behavior of complicated asynchronous circuits. The readers and writers example above could equally well describe a simple bus arbitration scheme. In fact, the finite-state assumption may be even more reasonable at the hardware level than at the monitor level.

Which brings us to the topic of this paper: What is the best way to translate path expressions into circuits? Lauer and Campbell have shown how to compile path expressions into Petri nets [6], and Patil has shown how to implement Petri nets as circuits by using a PLA-like device called an asynchronous logic array [11]. Thus, an obvious method for compiling path expressions into circuits would be to first translate the path expression into a Petri net and then to implement the Petri net as a circuit using an asynchrouous logic array. However, careful examination of Lauer and Campbell's scheme shows that a multiple path expression consisting of M paths each of length K can result in a Petri net with K^M places. Thus, the naive approach will in general be infeasible if the number of individual paths in a multiple path expression is large.

For the case of a path expression with a single path their scheme does result in Petri net which is comparable in size to the path expression. However, direct implementation of such a net using Patil's ideas may still result in a circuit with an unacceptably large area. An asynchronous logic array for a Petri net with P places and T transitions will have area proportional to $P \cdot T$ regardless of the number of arcs in the net. Since the nets obtained from path expressions tend to have sparse edge sets, this quadratic behavior may waste significant chip area.

Perhaps, the work that is closest to ours is due to Li and Lauer [8] who do indeed implement path expressions in VLSI. However, their circuits differ significantly from ours; in particular, their circuits are synchronous, and synchronization with the external world (which is, of course, inherently asynchronous) is not considered. Furthermore, their circuits use PLA's that result in an area complexity of $O(N^2)$. Rem [13] has investigated the use of a hierarchically structured path expression-like language for specifying CMOS circuits. Although he does show how certain specifications can be translated into circuits, he does not describe how to handle synchronization or give a general layout algorithm that produces area efficient circuits.

In contrast, the circuits produced by the construction described in this paper have area proportional to $N \cdot \log(N)$ where N is the total length of the multiple path expression under consideration. Furthermore, this bound holds regardless of the number of individual paths or the degree of synchronization between paths. As in [3] and [4] the basic idea is to generate circuits for which the underlying graph structure has a constant separator theorem [7]. For path expressions with a single path the techniques used by [3] and [4] can be adapted without great difficulty. For multiple paths with common event names, however, the construction is not straightforward, because of the potential need for synchronization at many different points on each individual path. Moreover, the actual circuits that we use must be much more complicated than the synchrouous ones used in ([3], [4]). Since events are inherently asynchrouous in our model, all of our circuits must be self-timed. This requires the use of special circuit design techniques and significantly complicates the proof that this circuit correctly captures the semantics of path expressions.

The paper is organized as follows: A formal semantics for path expressions in terms of partially ordered multisets [12] is given in section 2. In sections 3, 4, and 5 we give a hierarchical description of our scheme for implementing path expressions as circuits. In section 3 we first describe how the complete circuit interfaces with the external world. We then show how to build a *synchronizer* that coordinates the behavior of the circuits for the individual path expressions in a multiple path expression. In section 4 we describe a circuit for implementing single path expressions which we call a *sequencer*. In section 5 we show how the arbiter circuit used in section 3 can be implemented. We also argue that these circuits are correct and can be laid out efficiently. The paper concludes in section 6 with a discussion of issues such as fairness and of open problems such as the possibility of extending our construction to other synchronization mechanisms like the ones used in CCS and CSP.

2. The Semantics of Path Expressions

In this section we give a simple but formal semantics for path expressions in terms of partially ordered multisets of events [12]. We also relate our semantics to the one in terms of Petri Nets given by Lauer and Campbell [6].

Definition 1: A *partially ordered multiset* (pomset) over Σ is a triple $(Q, \leq, F)$ where $(Q, \leq)$ is a partially ordered set and F is a function which maps Q into Σ. □

An example of a pomset is shown in Figure 2-1. We use subscripts to distinguish different instances of the same element of Σ. Note that we could have alternatively defined a pomset as a directed acyclic graph in which each node is labeled with some element of Σ.

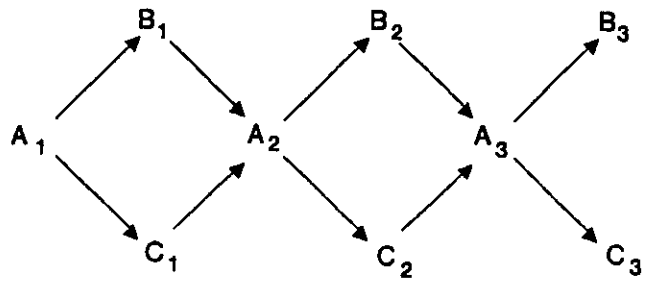

Figure 2-1: An example pomset

If the ordering relation of a pomset P over Σ is a total order, then we can naturally associate a sequence of elements of Σ with P; we will use S(P) to denote this sequence. In fact, a pomset should be regarded as a natural generalization of a sequence in which certain elements are permitted to be concurrent; this is why the concept is useful in modeling systems where several events may occur simultaneously.

Definition 2: If $P = (Q, \leq, F)$ is a pomset over Σ and $\Sigma_1 \subseteq \Sigma$, then the *restriction* of P to Σ_1 is the pomset $P\big|_{\Sigma_1} = (Q_1, \leq_1, F_1)$ where $Q_1 = \{d \in Q \mid F(d) \in \Sigma_1\}$ and $\leq_1$, F_1 are restrictions of $\leq$, F to Q_1, respectively. □

If P is a totally ordered pomset over Σ and $\Sigma_1 \subseteq \Sigma$, then $S(P\big|_{\Sigma_1})$ is just the *subsequence* of S(P) obtained by deleting all of those elements of Σ which are not in Σ_1.

A *simple path expression* is a regular expression with an outermost Kleene star. The only operators permitted in the regular expression are (in order of precedence) "*", ";", and "+". The "*" operator is the Kleene star, ";" is the sequencing operator, and "+" represents exclusive choice. Operands are event names from some set of events Σ that we will assume to be fixed in this paper. The outermost Kleene star is usually represented by the delimiting keyword **path ... end.** Thus $(a)^*$ would be represented as **path a end.**

A *multiple path expression* is a set of simple path expressions. As we will see shortly, each additional simple path expression further constrains the order in which events can occur. However, we cannot simply take as our semantics for multiple path expressions the intersection of the languages corresponding to the individual path expressions; two events whose order is not explicitly restricted by one of the simple path expressions may be concurrent. For example, in the multiple path expression for the readers and writers problem discussed in the introduction the two read events R_1 and R_2 may occur simultaneously. Nevertheless, we will still have occasion to use ordinary regular expressions in giving the semantics for path expressions; if R is an ordinary regular expression over Σ, then $\Sigma_R \subseteq \Sigma$ will be the set of symbols of Σ that actually appear in R and $L_R \subseteq \Sigma_R^*$ will be regular language which corresponds to R.

Definition 3: Let Σ be a finite set of events; a *trace* over Σ is a finite pomset $T = (Q, \leq, F)$ over Σ. We say that $i \in Q$ is an *instance* of an event $e \in \Sigma$ if $F(i) = e$. An instance i_1 of event e_1 *precedes* an instance i_2 of event e_2 if i_1 precedes i_2 in the partial order $\leq$. An instance i_1 of event e_1 is *concurrent* with an instance i_2 of event e_2, if it is not the case that i_1 precedes i_2 or that i_2 precedes i_1. □

In the example above A_1 precedes A_2, but B_1 and C_1 are concurrent.

Definition 4: Let R be a simple path expression with event set Σ_R. A trace T is *consistent with R* iff $T\big|_{\Sigma_R}$ is totally ordered and $S(T\big|_{\Sigma_R})$ is a prefix of some sequence in L_R. If M is a multiple path expression,

then a trace T is *consistent with M* iff it is consistent with each simple path expression R in M. $Tr_{\Sigma}(M)$ is the set of all traces which are consistent with M. □

Consider, for example, the multiple path expression M:

path A;B **end,**
path A;C **end.**

with $\Sigma = \{A, B, C\}$. It is easy to see that the trace in Figure 2-1 is consistent with each of the simple path expressions in M and hence is in $Tr_{\Sigma}(M)$.

3. Synchronizers for Multiple Path Expressions

This section describes our implementation of synchronizers for multiple path expressions. Figure 3-1 illustrates the interface between a synchronizer and the external world. Each event *e* is associated with a request line REQ_e and acknowledge line ACK_e. The synchronizer cooperates with the external world to ensure that these request and acknowledge lines follow a 4-cycle protocol:

1. The external world raises REQ_e to indicate that it would like to proceed with event *e*.
2. The synchronizer raises ACK_e to allow the external world to proceed with event *e*.
3. The external world lowers REQ_e, signifying completion of event *e*.
4. The synchronizer lowers ACK_e, signifying the end of the cycle and permission to begin a new one.

In this implementation, an event will occur during the period between cycles 2 and 3 in this protocol, where both REQ and ACK are high. Thus, multiple occurrences of any event *e* are non-overlapping in time, since any two occurrences are separated by the lowering of ACK and the raising of REQ.

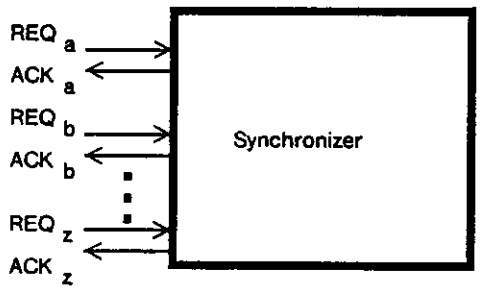

Figure 3-1: A synchronizer

An overview of a synchronizer circuit is shown in Figure 3-2. We describe below some of the building blocks in the circuit.

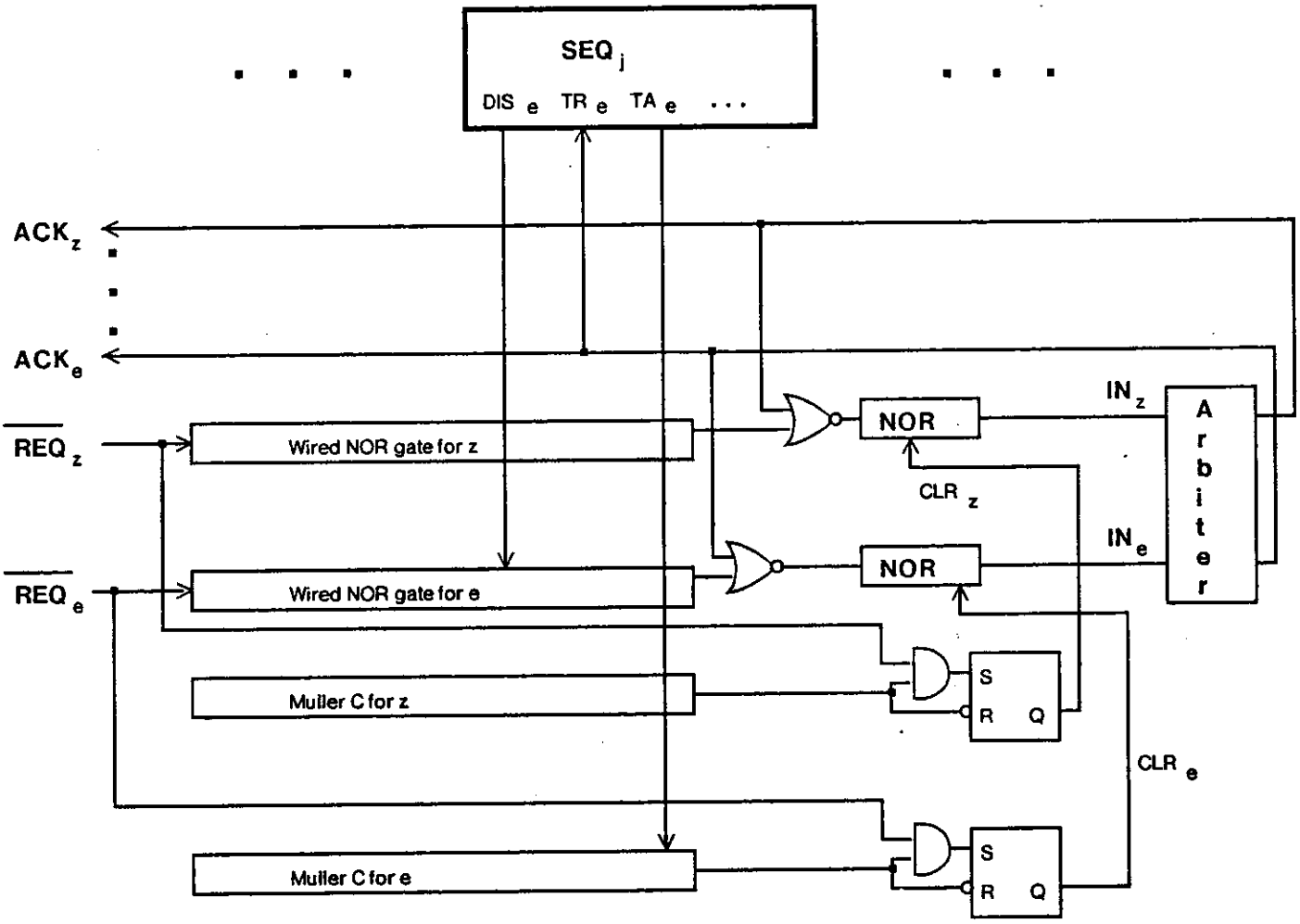

Figure 3-2: A synchronizer circuit

The C gate in Figure 3-2 is a Muller C-element; the output of a C-element remains low until all inputs are high and thereafter remains high until all inputs are low again. Its behavior then cycles. For an implementation see [14].

The arbiter in Figure 3-2 enforces pairwise mutual exclusion over the outputs corresponding to pairs of events which occur in the same path expression.In addition to enforcing mutual exclusion the arbiter tries to raise any output whose input is high. Most implementations of arbiters will have metastable states during which fewer signals than possible may be high at the output. Despite the metastable states, however, once an output signal has been raised, it remains high as long as the corresponding input remains high. The implementation of such an arbiter is discussed in detail in section 5.

Each sequencer block in Figure 3-2 ensures that the sequence of events satisfies one of the simple path expressions that comprise the multiple path expression. The synchronizer circuit contains one sequencer for each simple path expression, so that each simple path expression is satisfied by an execution event trace. For each event e that appears in a simple path, the corresponding sequencer has three connections: a request TR_e, an acknowledge TA_e, and a disable DIS_e. Events are sequenced by executing a 4-cycle protocol over one pair of the TR/TA lines. The DIS outputs of the sequencer are only valid between these cycles (when all TR and TA are low), and indicate which events would violate the simple path. The synchronizer will not initiate a cycle for any event whose DIS line is high. The implementation of the sequencer is given in section 4.

We now describe how the components of the circuit are interconnected. Refer to Figure 3-2. Let SEQ_e denote the set of sequencers for simple paths that contain event e. Every sequencer in SEQ_e has its DIS_e signal connected to a wired-NOR gate for e, its TA_e signal connected to a C gate for e, and its TR_e signal connected to ACK_e. The output of the latch at the end of the C gate for e, which is labeled CLR_e, is connected to each of the NOR gates in front of the arbiter which corresponds to event e or to some event mutually exclusive to e.

The following is an informal description of how the circuit works. The circuit behaves as shown in the timing diagram in Figure 3-3. When REQ_e is raised, event e is not allowed to proceed unless each sequencer in SEQ_e signals that at least one e type transition is enabled by negating DIS_e. Once this happens IN_e is raised, provided no mutually exclusive event is executing the second half of its cycle (and hence has its CLR high). If the arbiter decides in favor of some other pending event mutually exclusive to e, the above process repeats until e again gets a chance at the arbiter. Otherwise ACK_e will be raised and latched by the NOR gate arrangement in front of the arbiter. At this point the external world may proceed with event e. Simultaneously each sequencer in SEQ_e will find TR_e high and after some time raise TA_e. When all sequencers in SEQ_e have raised TA_e and the external world acknowledges completion of event e by lowering REQ_e, CLR_e will be raised. This causes ACK_e to be lowered. Each sequencer in SEQ_e will find TR_e low and after some time lower TA_e. When all such sequencers are done, CLR_E is lowered, and the cycle is completed.

To formally establish the correctness of our circuit , we must establish two things: First, we must show that the circuit allows only semantically correct event traces; second, that the circuit will allow any semantically correct event trace for some behavior of the external world. These properties of the circuit are often called *safeness* and *liveness* respectively. Our proof will make use of properties of the various circuit components shown in Figure 3-2. We list the most important of these properties as propositions, namely those relating to the sequencer, the arbiter, and the external world. Properties of other circuit components such as SR Flip-Flops, NOR gates, etc., are assumed to be well known and are used without further discussion. The proof also makes certain assumptions about the delays of the components:

1. The delay of the main NOR gate plus the 2-input NOR gate is less than that of the main Muller-C

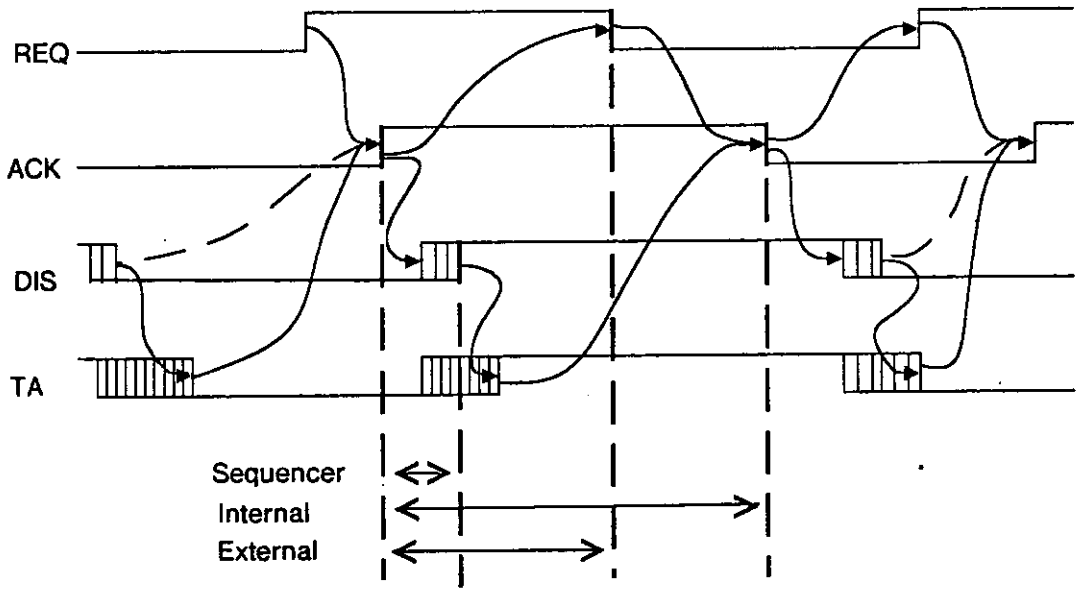

Figure 3-3: Synchronizer timing

element plus the SR Flip-Flop.

2. The maximum variation in delay for the NOR gates in front of the arbiter is less than the minimum delay of the arbiter.

We begin by introducing some notation that will be needed in the proof. Let the sequencers be denoted by SEQ_1 ... SEQ_p corresponding to the path expressions R1 ... Rp $\in$ M, and let Σ_{R1} ... Σ_{Rp} be the subsets of Σ that actually appear in R1 ... Rp respectively. Let I be a set of time intervals, which may include semi-infinite intervals extending from some finite instant to infinity. Each element in I is labelled by an element in Σ. Define T(I) to be the trace which has an element for each element in I and has the obvious partial order defined between elements whose time intervals are non-overlapping. Referring to Figure 3-3, let

- **Ext** = set of time intervals labelled 'external',
- **Int** = set of time intervals labelled 'internal',
- **Seq(j)** = set of time intervals labelled 'sequencer' for sequencer SEQ_j.

For every interval in **Int** with label *e* there are corresponding intervals with the same label in **Ext** and in every **Seq(j)** such that $e \in \Sigma_{Rj}$, namely those which start at the same time. We assume that the starting points of intervals in **Int** lie within some finite time period of interest, and the intervals in **Ext** and **Seq(j)** are restricted to intervals corresponding to those in **Int**.

With this notation in place we state some propositions, or axioms, that describe the properties of the circuit of Figure 3-2. These properties will be used to prove that the circuit is safe and live. The propositions that are not self-evident will be justified in later sections of this paper.

Proposition 5: (External world protocol): For all events *e*,

1. REQ_e is raised only if ACK_e is low.
2. REQ_e is lowered only if ACK_e is high. □

Proposition 6: (Arbiter safety and liveness):

1. For any events *e1*,*e2* that are mutually exclusive, ACK_{e1} and ACK_{e2} are never high simultaneously.
2. For any event *e*, ACK_e is raised only if IN_e is raised.
3. For any event *e*, ACK_e is lowered only if IN_e is low,and withing a of IN_e being lowered.
4. Consider a set of events $\Sigma' \subseteq \Sigma$, such that no two events in Σ' are in the same path expression. Then if all IN_e, $e \in \Sigma'$, are raised, within a finite time all ACK_e, $e \in \Sigma'$, will be raised. □

Proposition 7: (Sequencer protocol): For any sequencer SEQ_j,

1. TA_e is raised only if TR_e is high.
2. TA_e is lowered only if TR_e is low.
3. DIS_e is stable while all TR's and TA's in TR_e are low. □

Proposition 8: (Sequencer safety and liveness) : For any sequencer SEQ_j, assume that at all times,

- no two TR's are high simultaneously,
- TR_e is raised only if DIS_e and all TA's are low,
- TR_e is lowered only if TA_e is high.

Then the following hold :

1. TA_e is raised within a finite time of TR_e being raised.
2. TA_e is lowered within a finite time of TR_e being lowered.
3. For any sequencer SEQ_j, whenever all TA's and TR's are low, exactly those events *e* will have DIS_e low, for which S(T(Seq(j))) can be extended by *e* to give a prefix of some sequence in L_{Rj}. □

Proposition 9: (Initialization)

1. Sequencers are initialized with all TA's low.
2. The synchronizer circuit SR flip-flops are initialized to make all CLR's high. □

The following theorem states that a synchronizer satisfying Propositions 5 through 9 is provably safe.

Theorem 10: (Synchronizer Safety) : $T(\text{Ext}) \in Tr_{\Sigma}(M)$.

proof: See the appendix. □

As a converse to theorem 10 we would like to show that our circuit can produce any valid trace Ext, such

that T(Ext) ϵ $Tr_{\Sigma}(M)$ for at least some behavior of the external world. However for some traces T ϵ $Tr_{\Sigma}(M)$, there does not exist any Ext such that T(Ext)=T, so there is no way any circuit can produce the required trace Ext. This happens when T does not sufficiently constrain the order in which the elements may occur so that any actual set of time intervals will have fewer concurrent elements than T. Given such a T it is necessary to constrain its partial order relation further, by adding additional (consistent) precedence relationships. It is easy to show using definition 4 that this will never remove T from the set $Tr_{\Sigma}(M)$. We shall show that whenever T is sufficiently constrained so that it falls in a class of traces we call *layered*, then for some behavior of the external world T(Ext) for our circuit will equal this modified T.

Definition 11: A trace P = (Q,≤,L) is called *layered*, if Q can be subdivided into a sequence of *subsets*, such that for any *i1*, *i2* ϵ Q, *i1* precedes *i2* iff the *subset* in which *i1* lies precedes the *subset* in which *i2* lies. □

The trace in Figure 2-1 is layered, since its elements can be subdivided into the sequence of *subsets* $\{(A_1),(B_1,C_1),(A_2),(B_2,C_2),(A_3),(B_3, C_3)\}$ with the above property. If the size of each *subset* were one, then the trace would be totally ordered.

In general, any trace P will have a corresponding layered trace T which preserves most of the parallelism of P. It is easy to show that for any trace P,there exists a layered trace T, which differs from P only in that the partial order relation of P is a restriction of that of T.

Theorem 12: (Synchronizer Liveness): Given any layered trace P ϵ $Tr_{\Sigma}(M)$, our circuit will produce an event trace Ext, such that T(Ext) = P for some behavior of the external world. □

proof: See the appendix. □

4. Implementing the Sequencer for a Simple Path Expression

This section shows how to construct a sequencer that meets the conditions set forth in Propositions 7 and 8. The sequencer circuit is constructed in a syntax-directed fashion based upon the structure of the simple path expression. We show that a compact layout for the sequencer exists, so that circuits of this type can be implemented economically in VLSI.

Since a simple path expression is a regular expression, the sequencer for a simple path expression is similar to a recognizer for the regular expression. Although schemes for recognition of regular languages have been proposed that avoid broadcast [3], we will use a scheme that requires broadcast of events throughout the sequencer [4, 10]. Because our scheme for interconnecting sequencers requires broadcast, the broadcast

within an individual sequencer carries no additional penalty. A sequencer for a simple path expression is built up from primitive cells, each corresponding to one character in the path. The syntax of the path determines the interconnection of the cells in the sequencer. In this section, we first describe the behavior of a sequencer for a simple path expression, then give a syntax-directed construction method.

As noted in Section 3, a synchronizer communicates with each of its sequencers using three lines:

- TR_e: a signal to the sequencer that event *e* is about to commence in the external world;
- TA_e: an acknowledgement from the sequencer that all actions started by TR_e have completed;
- DIS_e: a status line indicating that action *e* would violate the path constraints so that TR_e should not be asserted.

These communication lines interact in a complex way. For a single type of event, the signals TR_e and TA_e follow the four-cycle signaling convention described in Section 3 for REQ and ACK. For different types of events, the synchronizer must guarantee the correct interaction of TR signals by ensuring that only one TR signal for an event satisfying the simple path expression is asserted at any time. The synchronizer can use the DIS status lines to determine which requests to send to the sequencer.

The sequencer also has a part to play in ensuring the correct interaction of TR, TA and DIS. Besides generating a TA signal that follows the four cycle convention with TR, it must ensure that the signal DIS_e is correct as long as no TR or TA signal is asserted. This guarantee means that if no TA is asserted, REQ_{e1} and REQ_{e2} are both asserted, and neither DIS_{e1} nor DIS_{e2} is true, then the synchronizer may choose arbitrarily between *e1* and *e2*, letting either of them through to the simple path sequencer. On receiving a TR_e signal, then, the sequencer must assert TA_e, adjust its internal state to reflect the occurrence of event *e*, assert the proper set of DIS lines, and await the negation of TR_e before negating TA_e.

Now that the behavior of a sequencer has been described, we show how to construct a sequencer for any path. A sequencer has two parts: a controller and a recognizer. The controller is connected directly to the rest of the synchronizer and generates both the TA signals and some control signals for the recognizer. The recognizer keeps track of which events in the path have been seen and generates the DIS signals.

Figure 4-1 shows the controller for a simple path P. The controller accepts the signals TR_e from the sequencer for each event *e* that appears in P. It generates the signals TA_e along with $Start_P$ and End_P. The meaning of TA_e is that all actions caused by TR_e have been completed. In this realization, TA is just a delayed version of TR, where the delay is long enough to let the sequencer stabilize. An upper bound on this delay can be computed from the layout of the rest of the circuit. It is possible to use a self-timed version of this circuit in

which the delay is derived from the recognizer. It has been omitted in this version of the paper as it unnecessarily complicates an understanding of how the circuits work. $Start_p$ and End_p are control signals that control the movement of data through the recognizer for P. $Start_p$ is true whenever at least one TR is on and no TA is on, while End_p is true whenever at least one TA is on and no TR is on.

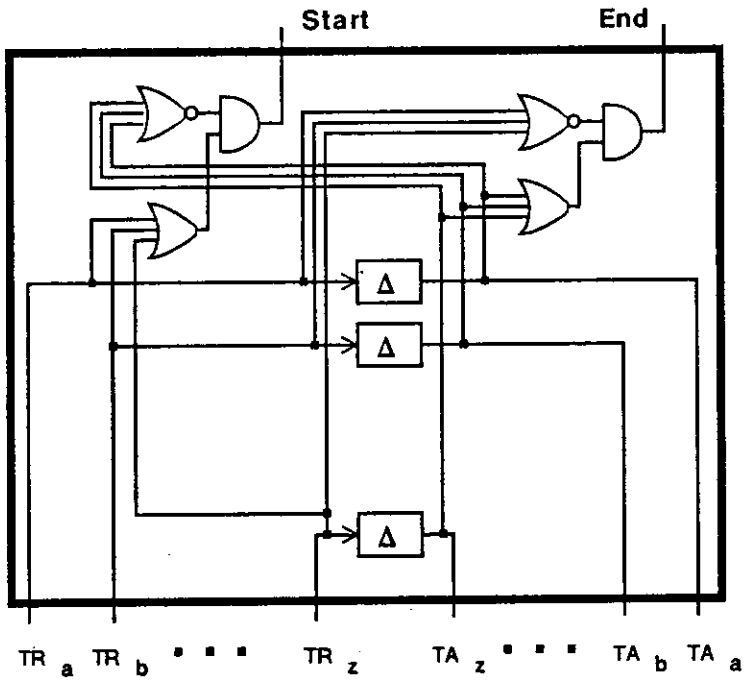

Figure 4-1: The controller for path P

The recognizer for a path accepts the TR_e signals and generates the DIS signals. It is made up of sub-circuits corresponding to subexpressions of the path. To construct the recognizer for a path, we parse the path using a context-free grammar. Productions that are used in parsing the path determine the interconnections of sub-circuits to form the recognizer. Non-terminals that are introduced in the parse correspond to primitive cells used in the circuit.

Recognizers are constructed using the following grammar for simple path expressions.

S → **path** R **end**
R → R;R | (R + R) | (R)* | <event>.

The terminal symbols in the grammar correspond to primitive cells; there is one type of cell for the "+" symbol, one for the "*" symbol, one for the ";" symbol, and one for each event. The non-terminals correspond to more complex circuits that are formed by interconnecting the primitive cells. Using the method described in [2], semantic rules attached to the productions of the grammar specify how the circuits on the right of each production are interconnected to form the circuit on the left.

To keep track of which events in the path have occurred and which are legal, the sub-circuits of a recognizer communicate using the signals ENB (enable) and RES (result). The circuit for a subexpression accepts ENB and uses it to determine when the first event in the subexpression is legal. It generates RES when the last event has occurred.

Figure 4-2 shows the cell for event *e*. Two latches, clocked by the signals $\mathbf{Start}_P$ and $\mathbf{End}_P$, control the flow of ENB and RES signals. Because of the definitions of $\mathbf{Start}_P$ and $\mathbf{End}_P$, the leftmost latch is loaded from ENB whenever at least one TR is on and no TA is on, while the rightmost latch is loaded to update RES whenever at least one TA is on and no TR is on. The two latches are never loaded at the same time; in fact, because TR and TA follow the four cycle signalling convention, there is a non-zero time between the end of the load signal for one latch and the start of the load signal for the other. Thus there is no combinational path through the cell.

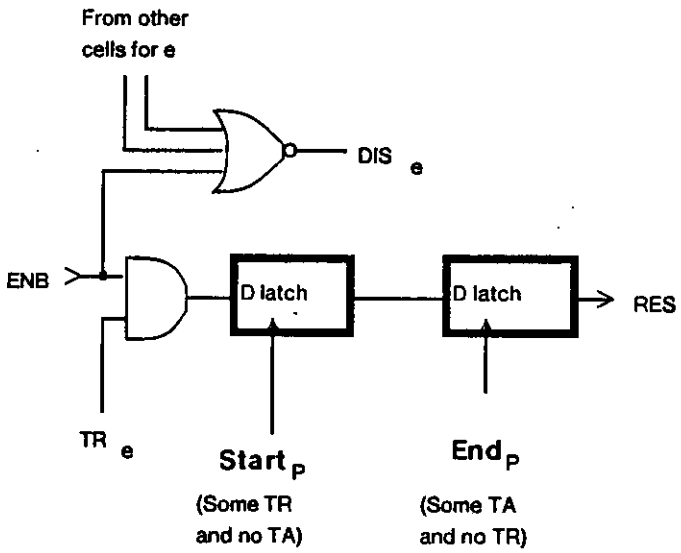

Figure 4-2: Cell for event *e* in path P

The event cell in Figure 4-2 propagates a 1 from ENB to RES only if event *e* occurs. When this cell is used in a recognizer for a path expression, the ENB input will be true if and only if event *e* is permitted by the expression. Thus, if ENB is true it negates DIS_e for the path, as shown in the figure. When a request TR is made, the output of the AND gate is loaded into the leftmost latch. If this request is TR_e, this output is 1; otherwise it is 0. In either case the output of the AND gate is propagated to RES through the latch when TR is lowered.

Figures 4-3 and 4-4 show the cells for the ";" and "+" operators. These are strictly combinational circuits. The circuit for ";" feeds the RES signal from the circuit at its left into the ENB signal for the circuit to its right.

The circuit for "+" broadcasts its ENB signal to its operands and combines the RES signals from its operands in an OR gate.

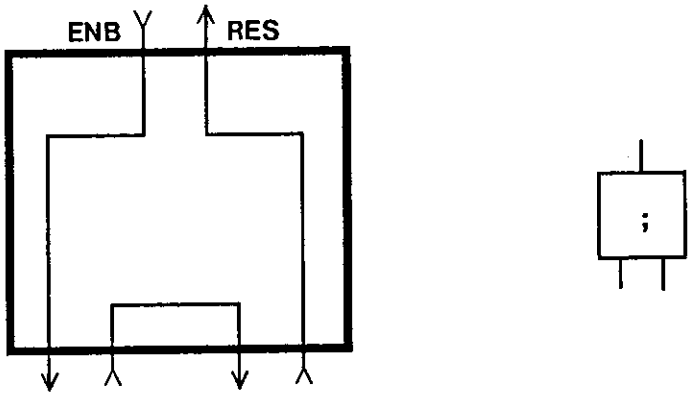

Figure 4-3: Cell for ";"

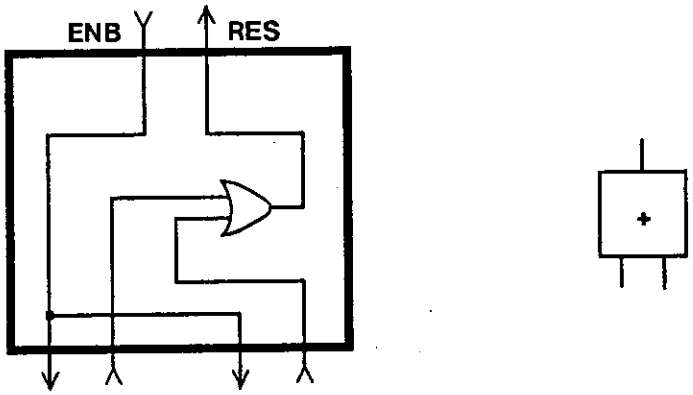

Figure 4-4: Cell for "+"

Figure 4-5 shows the cell for the "*" operator. The cell enables its operand after receiving either a 1 on either its own ENB or its operand's RES. Every time the operand is enabled the "*" cell also puts out a 1 on its own RES. It therefore outputs 1 on RES after 0 or more repetitions of its operand's expression. The additional AND gate sets the output to zero momentarily after each event, thereby preventing the formation of a latch when two or more "*" cells are used together or when the RES output is connected to the ENB input.

When larger circuits are made from these cells, the RES and ENB signals retain their meanings. Each event cell or sub-circuit formed from several cells accepts one input ENB and produces one output RES. We define ENB and RES to be correct if they meet the following conditions.

- ENB is true for a sub-circuit if each sequence of events satisfying the expression for the sub-circuit may be the next sequence to occur.

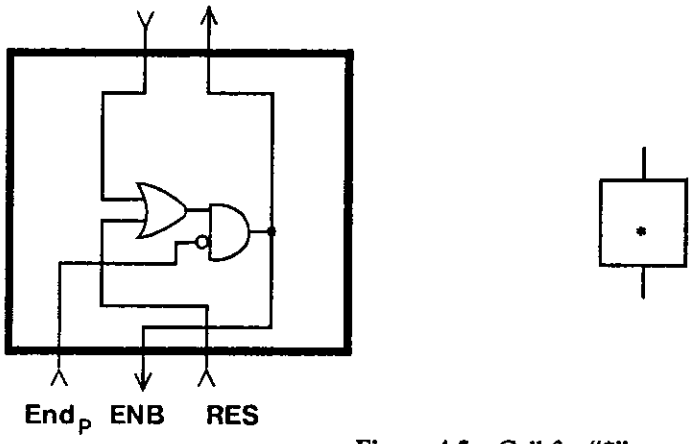

Figure 4-5: Cell for "*"

- RES is true for a sub-circuit if some sequence of events satisfying the sub-circuit has just occurred, and ENB was true before the beginning of that sequence.

The ENB and RES signals thus indicate that a subcircuit may start recognizing events, or that it has finished. In addition, a sequencer has a signal INIT, not shown in the figures, which clears the ENB inputs to all internal cells and sets the ENB inputs for the cells corresponding to the first events in the path.

The semantic actions for the productions of the grammar describe the interconnections of the cells in Figures 4-2, 4-3 and 4-4. Attributes are attached to the symbols of the grammar to represent the sets of events that appear in the path. These sets determine which TR and TA signals are combined to produce **Start$_p$** and **End$_p$**.

S[A] → **path** R[A] **end**
Hook the RES output of R to its ENB input, and connect INIT.

R[A ∪ B] → R[A];R[B]
Connect the RES output for R[A] to the ENB input of R[B]

R[A ∪ B] → (R[A] + R[B])
Connect the R's to the operand ports of a + cell.

R[A] → (R[A])* Connect R to the operand port of a * cell.

R[{*e*}] → event *e* Use a cell for *e* as the circuit for R

Figure 4-6 shows a recognizer for the path **path** a;(a+b);c **end** constructed using this syntax-directed technique.

All recognizers constructed by this procedure perform the correct function, as required by Propositions 7 and 8. That is, if a recognizer is initialized and some sequence of TR signals is sent to it, the recognizer will

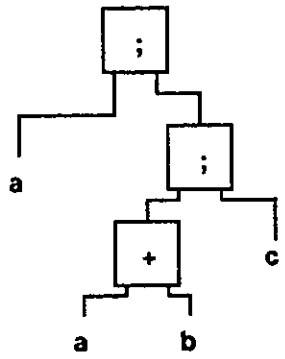

Figure 4-6: A recognizer for **path** a;(a+b);c **end**

output 1 on DIS_e for precisely those events *e* that are forbidden by the path. To prove this we show that the ENB input of an event cell in the recognizer is 1 if and only if the event corresponding to this cell is permitted by the path. As shown in Figure 4-2, DIS_e is 1 if and only if none of the cells for event *e* is enabled. Therefore, proving that an event cell has its ENB signal set if and only if the corresponding event is permitted in the path will show that the recognizer is functionally correct. In other words, we wish to prove that all ENB signals for event cells are correct, according to the definition of ENB above.

We shall prove the stronger statement that all ENB signals in the recognizer are correct. This proof is based upon the structure of the recognizer. An ENB signal in a recognizer is set by one of four sources:

- The operand port of a "+" or "*" cell;
- The left operand port of a ";" cell;
- The right operand port of a ";" cell;
- The INIT signal and the final RES of the recognizer;

In the first and second cases the signal is correct if and only if ENB for the operator cell is correct. In the third case the signal comes from the RES port of a recognizer for an initial subexpression. Therefore it is correct if and only if the RES signal for the subexpression is correct (asserted only at the end of the subexpression). In the fourth case the signal is correct at the start of recognition, and is correct thereafter if and only if the final RES signal is asserted only at the end of the expression. Thus, to prove that the circuits are correct, we need only prove that if the ENB signal for a recognizer is correct then so is the RES signal.

Once again, the proof of correctness is based upon the structure of a recognizer. In a correct recognizer the RES signal is true at time t_1 if and only if the ENB signal is true at some preceding time t_0 and the events between t_0 and t_1 obey the path. A recognizer that is a single event cell is clearly correct. A recognizer for

path a;b built by composition of correct subrecognizers for a and b is also correct, since if RES_b is true at time t_2 then there must be some time t_1 when RES_a was true, with all intervening events satisfying path b. But then there must have been a time t_0 when ENB_a was true and all events between t_0 and t_1 must satisfy path a. By definition of composition, then, the events between t_0 and t_2 satisfy a;b. A recognizer for path (a)* is correct if its subrecognizer is correct, since it outputs 1 and enables its operand if and only if ENB or RES_a is true. Finally, a recognizer for path a + b is correct if both subrecognizers are correct, since if RES is true then one of RES_a or RES_b must be true, and if one of ENB_a or ENB_b is true then ENB must be true. Since all methods of constructing recognizers have been shown to lead to correct circuits, recognizers constructed using this procedure are functionally correct.

Now that circuits have been designed and proved correct, we give compact layouts for them. The floorplan for a sequencer, shown in Figure 4-7 has the cells that make up the recognizer arranged in a line with the controller to one side. The TR signals flow parallel to the line of recognizer cells to enter the controller, and the **Start** and **End** signals emerge from the controller to flow parallel to the line of cells. The ENB and RES signals that are used for intercell communication also flow parallel to the line of cells.

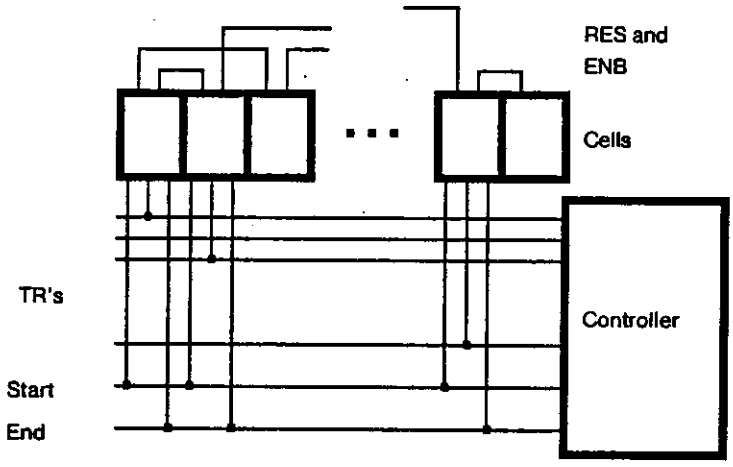

Figure 4-7: The floorplan for a sequencer

The layout in Figure 4-7 is fairly small. If the sequencer for a path of length n that has k types of input events is laid out in this fashion, the area of the layout is no more than $O(n(\log n + k))$. This is due to the structure of the recognizer circuits. All recognizer circuits are trees, which can be laid out with all nodes on a line and edges running parallel to the line using no more than $O(\log n)$ wiring tracks [7]. Thus the height of the circuit in Figure 4-7 is $O(\log n + k)$ while its width is $O(n)$.

5. Implementation of the Arbiter

In this section we briefly elaborate on the arbiter shown in Figure 3-2 to show that the conditions of Proposition 6 can be met. The main function of the arbiter is to select a single event from a mutually exclusive set of requests. Furthermore, the arbiter must be *fair* — any request that remains asserted must eventually be selected.

The following observation helps to simplify the arbiter: a pair of events occurring in any single path expression must be mutually exclusive. This is due to the role that each event plays in enforcing synchronization among a set of multiple path expressions that all contain the same named event. The arbitration function can thus be represented by a *conflict graph*, in which each event is denoted by a vertex and the relation between a pair of mutually exclusive events denoted by an undirected edge. Our observation shows that the resulting conflict graph for a set of path expressions consists of a set of overlapping cliques, where a clique of k nodes, A_1, A_2, ..., A_k, corresponds to a path expression R, with $\Sigma_R = \{ A_1, A_2, \ldots, A_k \}$. The conflict graph represents the static structure of a set of path expressions. Figure 5-1 shows a multiple path expression with its conflict graph.

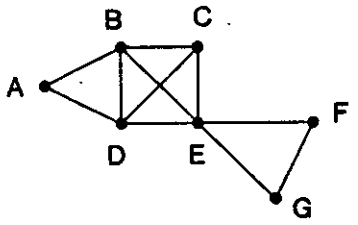

Figure 5-1: The conflict graph of a path expression

The dynamic behavior of the arbiter depends on the conflict graph together with the set of events that are enabled at any instant. The dynamic structure of the set of path expressions is represented by the subgraph of the conflict graph induced by the set of vertices corresponding to the events, enabled at that instant. The function of the arbiter is to select an independent set (not necessarily maximal) of this subgraph, thus ensuring that only one of any pair of mutually exclusive events is enabled.

Hence an arbiter is simply a transducer that takes a set of inputs and produces a set of outputs, subject to the constraints outlined earlier. Moreover, it is implicitly assumed that the arbiter is *oblivious* of any static or dynamic structure of the path expressions other than those represented by the conflict graph and the set of events enabled — in particular, it has no knowledge of the syntactic structure of the path expression, nor does it know the internal states of the individual sequencers. Clearly, one can build non-oblivious arbiters that may perform better, but this will be at the expense of conceptual simplicity and the area needed for additional logic and global wires.

To motivate our design we shall briefly discuss the problems with some simple schemes. In particular, we show that any deterministic oblivious arbiter gives rise to starvation of an event which is continually enabled. In similar vain, we show that a straight-forward extension of Seitz's scheme [14] for a two-input arbiter to a general conflict graph results in an unfair arbiter. Finally, we present a somewhat non-standard scheme implemented in CMOS which rectifies the problems with the other schemes.

The difficulty of building a fair deterministic arbiter can be illustrated by an example. Let $\Sigma = \{ A_1, A_2, \ldots, A_n \}$ be a set of events. To try to build a fair arbiter for Σ we might assign a priority number from 0 through $n - 1$ to each event, where the priority corresponds to the number of times the event is *blocked*, *i.e.*, the number of times the event is enabled but not selected by the arbiter. At any instant the arbiter selects from the set of enabled events with the highest priority number. When an enabled event is selected its priority number is reinitialized to the lowest value. On the other hand, if the enabled event is not selected its priority number is incremented by one. It seems that since an event A_i can have at most $n - 1$ neighbors in the conflict graph, and since each time it is blocked at least one of its neighbors is selected with a resulting increment in its own priority, after the n^{th} attempt A_i must have the highest priority among all the neighboring events and hence must be selected. However, an event may never be enabled even if its request is still pending because sequencing conditions imposed by the path expression may block the event. In order to make this observation concrete consider the following path expression:

path (A;C) + B;(A + B) **end.**

Assume that the external client always requests permission to perform all three events A, B and C. Let the priorities of all three be 0's initially. As a result, initially A and B are enabled. Assume that B is selected, making B's priority 0 and A's priority 1. In the next instant, A and B will again be enabled. But now A has the higher priority and will be selected, so that A's priority becomes 0 and B's becomes 1. Continuing in this fashion, it is easy to see that the sequence chosen will be B A B A B A .... The trouble with this scheme is that C will never be enabled even if its request is pending. This example can be extended to the following lemma.

Lemma 13: Let *M* be a deterministic finite-state transducer implementing an oblivious arbiter. Then there exists a path expression over $\Sigma = \{ A, B, C \}$ such that one event, say C, will be starved even though its request is continually pending.

Proof: Let *M* be a deterministic finite-state transducer whose alphabet is $\Sigma = \{ A, B, C \}$. Let the states of *M* be $S = \{ s_1, s_2, \ldots, s_m \}$. Let the conflict graph, *G*, for the path expression be the complete graph on the vertices A, B and C. We construct a path expression *P* with the conflict graph *G* such that *M*

causes the starvation of the event C. Notice that because of the nature of the conflict graph G, if at any instant A and B are enabled then at most one of A and B may be selected by M.

Let s_1 be an arbitrarily chosen state of M. We conduct an experiment on M by continuously providing A and B as the enabled inputs, starting with M in the state s_1. If we present a string of inputs { A, B }, { A, B }, ..., { A, B } of length m then we notice that at the 1st input { A, B }, the transducer deterministically goes from the state $s(1) = s_1$ to a state $s(2)$ while outputting A or B. Let $s(1)$, $s(2)$, ..., $s(m + 1)$ be the sequence of states and $\sigma \in \{ A, B \}^m$ be the output string produced as a result of the experiment. As a consequence of the pigeon-hole principle, some two states in the sequence of states will be the same . Of all such pairs, let $s(i)$ and $s(j)$ be two such states closest to s_1. Assume that $i < j$ and let k be the smallest multiple of $(j - i)$ such that $k \geq i$. Without loss of generality assume that M outputs B when in state $s(i)$ with the input { A, B }.

Let P be the path expression

$$\textbf{path } (A + B)^{i-1};(A;C + B); (A + B)^{k-i} \textbf{ end}$$

It is easy to see that P has G as the conflict graph and if the requests for A, B and C are continuously pending then the sequence of outputs will be a string in $\{ A, B \}^{\omega}$ and C will never be enabled. □

Before proceeding further, let us consider the path expression **path** A + B **end**, where the conflict graph is $G = (V, E) = (\{ A, B \}, \{[A, B]\})$. Seitz [14] has shown how to build an arbiter for such a structure using an interlock-element, as shown in Figure 5-2.

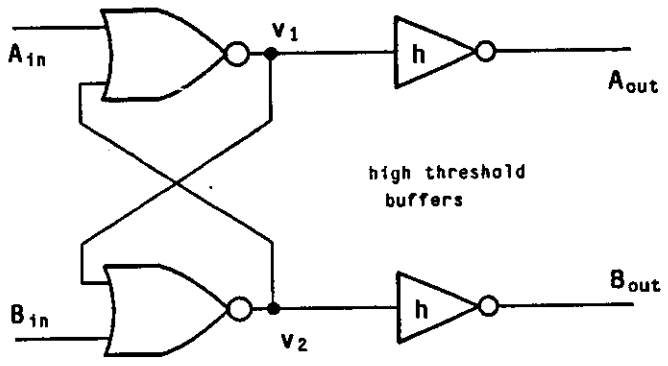

Figure 5-2: Seitz's Interlock Element

Circuit operation in Figure 5-2 is most easily visualized starting with neither client requesting, v_1 and v_2 both near 0 volts, and both outputs high. If any single input, say A_{in}, is lowered then v_1 is driven high, resulting in A_{out} being lowered — B_{out} remains unaffected. Moreover, once A_{out} is lowered, and as long as A_{in} is kept low, the interlock element remains in this stable state irrespective of what happens to B_{in}. If A_{in} is now raised high, then the element returns to its initial condition if B_{in} is still high; or B_{out} is lowered if B_{in} is lowered in the meantime.

However, the interesting situation occurs when both A_{in} and B_{in} are both lowered concurrently or within a very short interval of time. In this case the cross-coupled NOR gates enter a metastable state, which is resolved after indeterminate period of time in favor of either A or B. Since this resolution depends on the thermal noise generated by the gates, it is inherently probabilistic. In this case the outputs of the NOR gates themselves cannot be used as the outputs. High threshold inverters between the NOR gates and the outputs prevent false outputs during the metastable condition.

It would seem natural to extend Seitz's idea by generalizing it to the conflict graph for an arbitrary set of path expressions. Roughly speaking, we may construct a circuit by homomorphically transforming the conflict graph to a circuit by replacing each vertex with a NOR gate and each edge with a cross-coupling of NOR gates corresponding to the pair of vertices on which the edge is incident. However, such an implementation in NMOS has some severe problems, which will be clarified if we consider the circuit for the readers-writers path expression:

path R_1 + W **end**
path R_2 + W **end**

where the pair R_1 and W and the pair R_2 and W are mutually exclusive. The conflict graph and the circuit for this expression are shown in Figure 5-3.

Consider the situation when the circuit is in the none-requesting condition and all three requests, R_1, R_2 and W, arrive concurrently. An infinitesimally short interval Δt after all three requests arrive, let us assume that the voltages at the outputs (of the NOR gates) have increased by an infinitesimally small value $\Delta v \ll v_{th}$. The pull-down MOS transistors may be assumed to be operating in their linear region. If all pull-ups are assumed to provide equal active resistance, the output of the NOR gate corresponding to W will grow less rapidly than those corresponding to R_1 or R_2. The cumulative effect of this imbalance will result in a low output for W's NOR gate and high outputs for R_1's and R_2's. Hence if R_1, R_2 and W request continuously then the request for W will never go through, resulting in W's starvation.

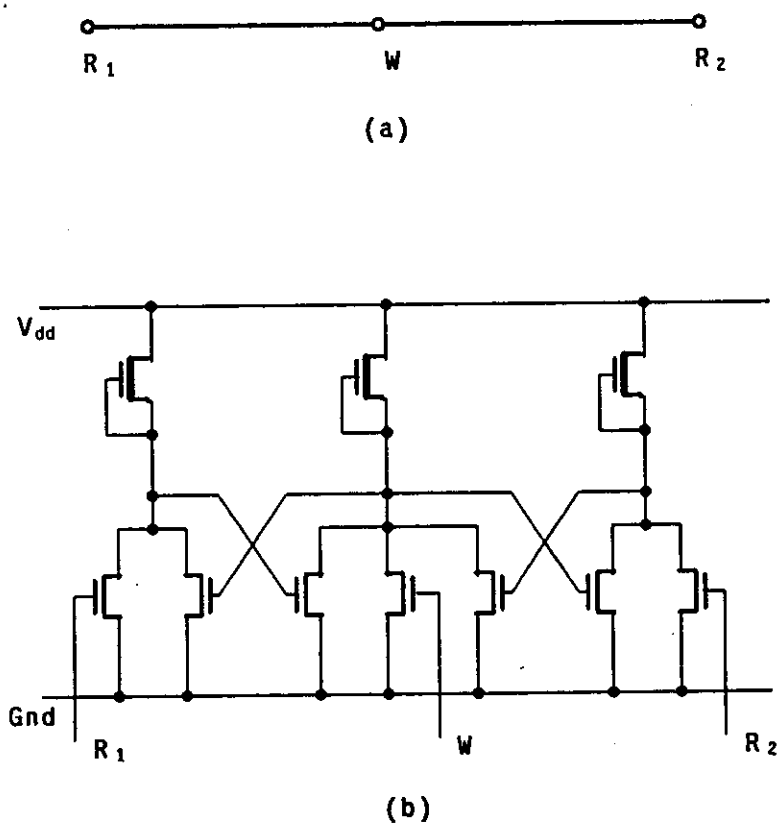

Figure 5-3: (a) The Conflict Graph and (b) The Arbiter in NMOS.

An easy fix to this problem may be to increase the ratio of pull-up to pull-down for W's NOR gate to twice that of R_1's and R_2's. But if this is done in a static manner then, when only R_1 and W are requesting, W will have an unfair advantage over R_1. Obviously, what is needed is some means of controlling the ratios such that depending on the set of requests the circuit configures itself dynamically in order to behave in a balanced fashion.

An arbiter that can configure itself dynamically for the problem with two readers and one writer is shown in Figure 5-4. To see how this scheme remedies the problem discussed earlier, consider the situation when the circuit is in non-requesting condition and all three requests, R_1, R_2 and W, arrive concurrently. An infinitesimally short interval Δt after all three requests arrive, the voltages at the outputs will have increased by an infinitesimally small value $\Delta v \ll v_{th}$. The pull-down MOS transistors are in their linear region. However, since active resistances of the pull-up transistors depend on the neighboring events that are enabled,

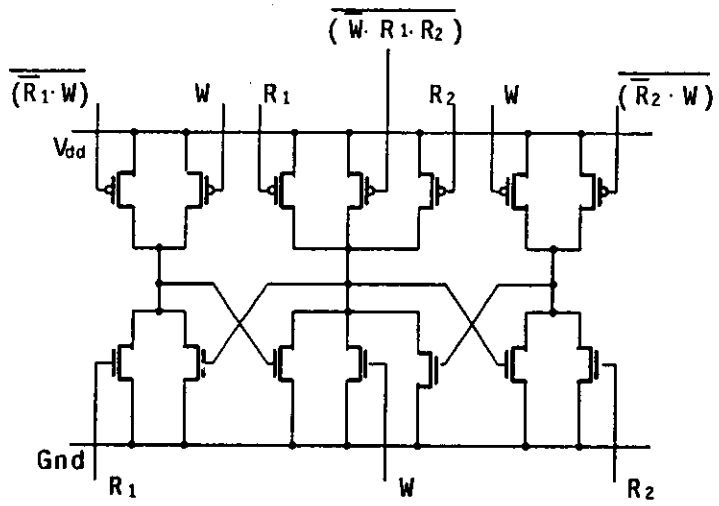

Figure 5-4: The Arbiter for 1-Writer-2-Readers Problem in CMOS.

the pull-up resistance of the gate associated with W is exactly half of that associated with R_1 or R_2. This provides a balance among pull-up resistances and results in almost equal rate of growth of voltages at the outputs. Hence the interlock elements enter their metastable states more or less simultaneously; and the metastable condition is resolved either in favour of R_1 and R_2 or in favour of W, the choice governed by statistical thermal phenomena.

A similar analysis shows that the circuit behaves correctly when only two out of three requests arrive concurrently. However, if only one request, say W, arrives while all its neighbours remain in their non-requesting condition the circuit behaves somewhat differently. In this case the pull-up transistor with input $(\overline{W \cdot R_1 \cdot R_2})$ will turn on, thus allowing the output of the gate to go high. It is important to observe that the pull-up transistors are controlled dynamically by the requests for the neighbouring events — if there is a request for the neighbouring event then only the pull-up corresponding to the event turns on; and if there is no request for the neighbouring events then only the pull-up corresponding to the event itself turns on. For this to be implemented correctly it is essential that the pull-up corresponding to the event itself be turned on only after a delay necessary for the requests for the neighbouring events to propagate to the gate of the pull-up.

The complex statistical nature of thermal noise in the circuit in conjunction with the complexity of the structure of the conflict graph makes it hard to analyze the circuit electrically. For instance, the time constants associated with each arbiter output could possibly differ significantly. Under the assumption that these second order effects are small, every enabled event will have a positive non-zero probability of being selected. Thus, for a reasonable class of path expressions, the circuit ensures that a continuously requesting event is eventually selected. This class includes the path expressions for which the other two arbiters can not provide a

good solution.

6. Conclusion

So far we have not discussed fairness. Intuitively, the implementation of a path expression is *fair* if any continuously requesting event will be eventually selected, provided it is possible to do so without violating the semantics of the path expression. As pointed out in the previous section, our implementation is fair for a reasonable class of path expressions. As an example of a path expression for which our implementation is not fair consider the following :

path (A + B); C **end,**
path D; (A + E) **end**

Suppose that each event takes the same amount of time to execute externally and that new requests for each event are forthcoming as soon as allowed by the protocol. Then simultaneous execution of D and B will alternate with simultaneous execution of C and E without the arbiter ever having to block any event. Yet, event A will never execute even if it remains continually ready. If, however, the first request for event B is delayed by the time it takes to execute an event, then initial execution of event D may be followed by alternate executions of A and (D,C). Since neither the duration of external events nor the occurrence of external requests is under the control of the circuit, it is not easy to ensure fairness for such path expressions. It remains an open question whether a practical solution to this problem exists.

Since our circuits have the constant separator property, a more compact O(N) layout is be possible using the techniques of [4]. However, while it is definitely possible to automatically generate the $O(N \cdot \log(N))$ layout that we propose, it is much more difficult in practice to generate the O(N) layout of [4]. Furthermore, the O(N) layout will occupy less area only for very large N. We suspect that ease of generating the layout will win over asymptotic compactness in this case.

Finally, we plan to investigate extensions of our construction to appropriate finite state subsets of CSP [5] and CCS [9]. In the case of CSP the subset will only permit boolean valued variables and messages which are signals. If the number of message types is fixed, we conjecture that area bounds comparable to those in section 4 can be obtained. Arrays of processes in which the connectivity of the communication graph is low can be treated specially for a more compact layout. Such a finite-state subset of CSP may even be more useful than the path expression language discussed in the paper for high level description of various asynchronous circuits.

References

1. Campbell, R. H. and A. N. Habermann. The Specification of Process Synchronization by Path Expressions. In *Lecture Notes in Computer Science, Volume 16*, G. Goos and J. Hartmanis, Ed.,Springer-Verlag, 1974, pp. 89-102.

2. Foster, M. J. *Specialized Silicon Compilers for Language Recognition.* Ph.D. Th., CMU, July 1984.

3. Foster, M. J. and Kung, H. T. "Recognize Regular Languages with Programmable Building-Blocks." *Journal of Digital Systems VI*, 4 (Winter 1982), 323-332.

4. Floyd, R. W. and Ullman, J. D. "The Compilation of Regular Expressions into Integrated Circuits." *Journal of the Association for Computing Machinery 29*, 3 (July 1982), 603-622.

5. Hoare, C. A. R. "Communicating Sequential Processes." *Comm. ACM 21*, 8 (1978).

6. Lauer, P. E. and Campbell, R. H. "Formal Semantics of a Class of High-Level Primitives for Coordinating Concurrent Processes." *Acta Informatica 5* (June 5 1974), 297-332.

7. Leiserson, C.E. *Area-Efficient VLSI Computation.* Ph.D. Th., Carnegie-Mellon University, 1981.

8. Li, W. and P. E. Lauer. A VLSI Implementation of Cosy. Tech. Rept. ASM/121, Computing Laboratory, The University of Newcastle Upon Tyne, January, 1984.

9. Milner, Robin. *A Calculus of Communicating Systems.* Volume 92: *Lecture Notes in Computer Science.* Springer-Verlag, Berlin Heidelberg NY, 1980.

10. Mukhopadhyay, A. "Hardware Algorithms for Nonnumeric Computation." *IEEE Transactions on Computers C-28*, 6 (June 1979), 384-394.

11. Patil, Suhas S. An Asynchronous Logic Array. MAC TECHNICAL MEMORANDUM 62, Massachusetts Institute of Technology, May, 1975.

12. Pratt, V. R. On the Composition of Processes. Symposium on Principles of Programming Languages, ACM, January, 1982.

13. Rem, Martin. *Partially ordered computations, with applications to VLSI design.* Eindhoven University of Technology, 1983.

14. Seitz, C. L. "Ideas About Arbiters." *LAMBDA First Quarter* (1980), 10-14.

Appendix : Proof details

Refer to section 3:

Lemma 14: If the same assumptions as in proposition 8 are satisfied, then T(Seq(j)) is consistent with R_j.

Proof: From proposition 8 it follows that Seq(j) consists of non concurrent time intervals. The result is therefore easy to prove by induction on the number intervals in Seq(j), using the same proposition. □

Lemma 15: For each element *i* in Int with label *e*, the corresponding elements in Ext and Seq(j) are subintervals of *i*.

Proof: (requires proof based on the properties of the circuit in fig 3-2). □

Lemma 16: For any Rj ∈ M, $T(Int)|_{\Sigma_{Rj}}$ is a totally ordered multiset.

Proof: It is easy to show that $T(Int)|_{\Sigma_{Rj}} = T(Int|_{\Sigma_{Rj}})$. But $Int|_{\Sigma_{Rj}}$ consists of 'internal events' of the path expression Rj, during each of which the corresponding ACK is high. Hence by proposition 6, no two such events overlap, and therefore $T(Int)|_{\Sigma_{Rj}}$ is a totally ordered multiset. □

Lemma 17: For any Rj ∈ M, $T(Int)|_{\Sigma_{Rj}} = T(Ext)|_{\Sigma_{Rj}}$.

Proof: For any element *i* of T(Int), that is also in $T(Int)|_{\Sigma_{Rj}}$, the corresponding element of T(Ext) will be in $T(Ext)|_{\Sigma_{Rj}}$ (definition 2) since they must map to the same alphabet $e \in \Sigma_{Rj}$. Hence these traces have the same number of elements. Also from lemma 15 it follows that if *i1* and *i2* are two elements of $T(Int)|_{\Sigma_{Rj}}$ satisfying one or none of "*i1* precedes *i2*" and "*i2* precedes *i1*", the corresponding elements of $T(Ext)|_{\Sigma_{Rj}}$ will satisfy at least the same relationships. In other words the partial order of T(Int) is a restriction of that of T(Ext). But by lemma 16 $T(Int)|_{\Sigma_{Rj}}$ is a totally ordered multiset. Hence from the above $T(Ext)|_{\Sigma_{Rj}}$ will have the same partial order relationship and, therefore, be the same totally ordered multiset. □

Lemma 18: For any Rj ∈ M, $T(Seq(j)) = T(Int)|_{\Sigma_{Rj}}$.

Proof: Follows from lemma 15 and 16 in the same way as in the proof of lemma 17. The only difference is that $T(Seq(j))|_{\Sigma_{Rj}} = T(Seq(j))$. □

Lemma 19: For any sequencer SEQ_j, no two TR's are high simultaneously.

Proof: The two TR's would be two ACK's of events in the same path expression Rj, which cannot be high simultaneously by proposition 6. □

Lemma 20: For any sequencer SEQ_j, TR_e is raised only if DIS_e is low and all TA's are low.

Proof: By induction on the number of rising transitions of TR's :

1. (First transition): Let the corresponding event be *e*. By proposition 9 initially all TA's are low, and all CLR's are high, hence all TR's are low initially. By proposition 7 all TA's will remain low until the first rising transition of TR_e. By the same proposition DIS_e will not change until the first rising transition of TR_e. If DIS_e were not low, IN_e would remain low (see Figure 3-2). Hence by proposition 6, TR_e would remain low, a contradiction.
2. (For a succeeding transition): Let the corresponding event be *p* and that of the previous transition *q*. While TR_q is high no TA or TR other than TA_q or TR_q can be high (proposition 6 and lemma 19). Until CLR_q goes high, TR_q must remain high (see Figure 3-2). Once CLR_q goes high, all IN_a, with $a \in \Sigma_{Rj}$, will be low after a short delay (see Figure 3-2). Assuming the variation in this delay for different *a*'s is less than the delay of the arbiter in lowering TR_q, all TR_a with $a \neq q$ will continue to

remain low until CLR_q is lowered (see Figure 3-2). All TA_a, with $a \neq q$, also continue to remain low (proposition 7). But CLR_q remains high at least until TA_q is lowered (see Figure 7). Hence by the time TR_p is raised all TA's will be low. Also TR_p could not have been raised if IN_p were low (proposition 6). But if DIS_p was high when TA_p was last lowered then IN_p would now be low (see Figure 3-2), assuming the main NOR gate plus the 2-input NOR gate have a lesser delay than the Muller-C element plus the SR Flip-Flop. Moreover, DIS_p cannot change before TR_p is raised (proposition 7). Hence DIS_p must be low when TR_p is raised.

□

Lemma 21: For any sequencer SEQ_j, TR_e is lowered only if TA_e is high.

Proof: The NOR gate arrangement in front of the arbiter insures that once TR_e is high it remains high until CLR_e is raised, and this can occur only if TA_e is high (see Figure 3-2). Moreover once TA_e is high it will remain high until TR_e is lowered (proposition 7). □

Theorem 10

Proof: Lemmas 19,20,21 satisfy the preconditions of proposition 8. Hence T(Seq(j)) is consistent with Rj for any Rj ∈ M. By lemma 18 and definition 4, T(**Int**) is consistent with Rj for any Rj ∈ M. By lemma 17 and definition 4, T(**Ext**) is consistent with Rj for any Rj ∈ M. Hence by definition 4, T(Ext) ∈ $Tr_\Sigma(M)$.

□

Lemma 22: If T ∈ $Tr_\Sigma(M)$ is layered, then each *subset* (cf definition 11) of T has the property that no two elements in it are instances of events in Σ_{Rj} for any Rj ∈ M.

Proof: Any two elements *i1,i2* (corresponding to events *e1,e2*) in the same *subset* of T must be concurrent (definitions 3,11). Suppose *e1,e2* ∈ Σ_{Rj} with Rj ∈ M. Then $T|_{\Sigma_{Rj}}$ will include *i1,i2* which will be concurrent (definition 2). Hence $T|_{\Sigma_{Rj}}$ cannot be a total order and therefore T ∉ $Tr_\Sigma(M)$ (definition 4) -- leading to a contradiction. Hence the result. □

Theorem 12

Proof: The behavior we require of the external world is that it simultaneously raise REQ for all events in the first *subset* of T, wait until all corresponding ACK are high, then simultaneously lower all REQ, wait until all ACK are low, then repeat this *cycle* for the next *subset* of T, and so on. We need to show that under these conditions the circuit responds within a finite amount of time in each *cycle*. The result then follows directly.

As shown in the proof of lemma 20, all ACK's are initially low. Hence they are low at the beginning of

each of the *cycles* mentioned in the previous paragraph. At the beginning of each such *cycle*, Ext,Int and every Seq(j) with Rj ∈ M, get redefined. Let Tp denote T restricted to subsets before the current cycle. It is easy to show by induction on the number of cycles and definition 4 that at the beginning of each cycle T(Ext) = Tp and Tp ∈ $Tr_{\Sigma}(M)$. Hence for any Rj ∈ M, $S(Tp|_{\Sigma_{Rj}})$ is a prefix of some element in L_{Rj}. If the next *subset* contains an instance *il* of event *el*, then for each Rj ∈ M such that *el* ∈ Σ_{Rj}, $S(Tp|_{\Sigma_{Rj}})$ can be extended by *il* to give a prefix of some sequence in L_{Rj} ; in fact this extension gives the next value of $Tp|_{\Sigma_{Rj}}$ (see lemma 22). But by lemmas 18,17, for any Rj ∈ M, $T(Seq(j)) = T(Ext)|_{\Sigma_{Rj}} = Tp|_{\Sigma_{Rj}}$. Hence for each Rj ∈ M, such that el ∈ Σ_{Rj}, T(Seq(j)) can be extended by *il* to give a prefix of some sequence in L_{Rj}. Thus by proposition 8, the corresponding sequencers SEQ_j, with *el* ∈ Σ_{Rj}, will have DIS_j low. This applies to any *el* in the next *subset* of T.

Therefore at the beginning of any cycle, when REQ_{el} for any event *el* in the next subset of T is raised, all DIS_{el} inputs to the NOR gate for event *el* (see Figure 3-2), will be low. Also within a finite amount of time all relevant TA_{el}'s must go low by proposition 8, since the corresponding TR_{el}'s are already low. Hence CLR_{el} will go low, and IN_{el} will go high for each *el* in the next subset of T. It follows from proposition 6 and lemma 22 that all ACK's corresponding to events in the next *subset* of T will be raised within a finite amount of time.

The proof for the second half of the cycle is more straightforward. By lemma 8 once all REQ's are lowered, within a finite time all relevant TA's will be raised, causing the corresponding CLR's to go high. As a result all relevant IN's go low (see figure 3-2) and hence by proposition 6 all ACK's go low within a finite time, completing the cycle. □