Miss Manners: A Specialized Silicon Compiler for Synchronizers

T. S. Balraj
M. J. Foster
Department of Computer Science
Computer Science Building
Columbia University
New York City, 10027

CUCS-185-85

Abstract

Miss Manners is a *synchronizer generator* that will produce the layout of a synchronizer given a high-level description. A synchronizer generator is a type of specialized silicon compiler. Synchronizer generators can greatly aid the design of systems that are structured as loosely-coupled networks of autonomous subsystems. Chips that are structured in this way have reduced communication requirements and greater tolerance for transient failures. We describe a language for specifying synchronization requirements and a compiler for translating the language into circuits that enforce the specifications.

1 Introduction

VLSI circuits that are structured as loosely-coupled networks of autonomous subsystems have many advantages over monolithic systems. The major advantage is locality of communication. By performing most of the computation within small areas of a chip, and communicating over long distances only when absolutely necessary, VLSI systems can avoid many of the costs due to long distance communications. Large chips that are not structured in this way, but are monolithic, can require large volumes of data to be sent long distances, which requires the use of large drivers and wires with large delays. The advantages of loosely-coupled circuits over monolithic ones are similar to the advantages of distributed computing systems over conventional multiprocessors. In addition to reduced communication requirements, the benefits include increased reliability and extensibility.

Loosely-coupled VLSI circuits can be locally synchronous but globally self-timed. The autonomous subcircuits of a loosely-coupled system can each have their own clocks, which can be independent of each other. The occasional communications between autonomous subcircuits can be self-timed, using stretchable clocks or other mechanisms [5, 17] to avoid synchronizer failure. This design approach avoids clock skew and the other problems of clock distribution that are exhibited by large clocked systems. At the same time it permits compact, easily understood synchronous circuits to be used for most of the chip. VLSI systems designed in this way can thus have faster-running clocks and be more reliable than strictly synchronous systems, while being more compact and easier to design than strictly self-timed systems.

Although the use of loosely-coupled architectures is beneficial, it places some requirements on VLSI circuits. One requirement is synchronization between subcircuits. Systems of this type may require mutually exclusive access to communication channels, as well as more complex types of synchronization [6]. An "etiquette authority" that enforces the synchronization requirements must be included in the system. Furthermore, if the system has several subcircuits with independent clocks, some means of communication between the subcircuits must be devised that avoids synchronizer failure [4]. The etiquette authority must aid the subcircuits in communicating, and should be capable of asynchronous communication with all other subcircuits.

How should this etiquette authority behave? It interacts with the other subcircuits of the VLSI system to ensure harmonious cooperation. The subcircuits must be well-mannered: when a

subcircuit wishes to perform some action (such as communication on a shared channel) that might be in violation of the synchronization requirements, it first asks for permission from the etiquette authority. The subcircuit then waits until the etiquette authority gives it permission to proceed. The etiquette authority must examine all actions that are proceeding and withhold its permission until the requested action is safe.

The etiquette authority, or synchronizer, may be a complex circuit. It must communicate asynchronously with its client processes, and must keep track of synchronization conditions that could be arbitrarily complicated. To make the loosely-coupled design style feasible in practice, we must develop synchronizer generators for VLSI. These generators will accept the specification of a synchronization structure as input and produce a layout of the required synchronizer in the technology to be used for the chip. The use of synchronizer generators eases the task of fulfilling the requirements of loosely-coupled VLSI systems. With a synchronizer generator, circuits can be designed in this beneficial style without excessive design time.

Our synchronizer generator, Miss Manners, is a specialized silicon compiler, which is a collection of cells and interconnection rules for generating layouts within a restricted application area [8]. Specialized silicon compilers translate behavioral specifications into efficient circuits. The use of behavioral specifications ensures that the compiler will be easy to use, while the restriction of the application area of the compiler ensures that it can create efficient circuits. Knowledge of an application area can be built into a specialized silicon compiler to aid in constructing small, fast circuits; a more general compiler must rely on more general, less effective optimization techniques.

Miss Manners uses *path expressions* to specify the behavior of synchronizers. Path expressions were originally proposed by Campbell and Habermann [3] for synchronization within software systems. For example, the synchronization required for a file with two reader processes and one writer process may be specified by the following multiple path expression:

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

The first path prevents a read operation by process 1 from occurring at the same time as a write operation; the second path similarly restricts process 2. The path expression does not restrict the two reads from occurring simultaneously, but a read and write cannot occur at the same time.

Path expressions are useful in describing the synchronization required in cooperating circuits, because they provide a combination of simplicity and expressive power. Because path expressions are closely related to regular expressions, their semantics are easily understood. At the same time, they have the power to describe many of the synchronization needs of VLSI circuits. For example, the readers and writers example above could describe a simple bus arbitration scheme. As long as the synchronization required by a system is finite-state, path expressions are an appropriate description of it.

Miss Manners translates a path expression into a synchronizer circuit. Some work on this problem has been done previously. For example, Lauer and Campbell have shown how to

compile path expressions into Petri nets [11] and Patil as shown how to implement Petri nets using an asynchronous logic array [15]. One method for compiling path expressions into circuits would thus be to first convert the expression into a Petri net and then to implement the Petri net as a circuit using an asynchronous logic array. However, this may produce a circuit that is much larger than necessary. A multiple path expression consisting of k paths, each of length n, can result in a Petri net with n^k places if Lauer and Campbell's scheme is used. This plan may therefore be infeasible for implementing a path expression with a large number of individual paths.

Even for a path expression with a single path, this implementation may lead to a circuit that is too large. Although the Petri net produced by Lauer and Campbell's translation is relatively small in this case, the asynchronous logic array produced by Patil's implementation may be large. The array for a Petri net with p places and t transitions has area proportional to pt, regardless of the number of arcs in the net. Since path expressions tend to produce nets with sparse edge sets, this quadratic behavior may waste chip area.

Li and Lauer [13] have presented an implementation of path expressions in VLSI. Their circuits have some drawbacks, however. First, the circuits are synchronous; asynchronous inputs from the external world, or from distant parts of the chip, are not considered. Second, their circuits use PLA's that can result in area $O(n^2)$ for a path expression of length n.

Miss Manners is based upon the translation method of Anantharaman *et al.* [1]. The circuits described there have several advantages in loosely-coupled VLSI systems. First of all, the circuits produced by our construction have area at most $O(n \log n + v^2)$ for a multiple path expression with v variables and total length n. Moreover, the circuits are asynchronous; events do not have to be synchronized to a global clock. A self-timed arbiter is employed within the circuit to resolve conflicts between competing events.

2 Describing Synchronizers

A synchronizer interacts with *clients*, which are other subcircuits of the system, to schedule *events*. Events correspond to actions that the clients might take that could interfere with other actions. The synchronizer will ensure that the events occur in a legal order. As shown in Figure 1, each event corresponds to two lines connected to the synchronizer. One line, REQ is controlled by the client and read by a synchronizer; the other ACK is controlled by the synchronizer and read by the clients. The synchronizer cooperates with its clients to ensure that these request and acknowledge lines follow a four-cycle protocol:

1. A client raises REQ_e to indicate that it would like to proceed with event e.
2. The synchronizer raises ACK_e to allow the client to proceed with event e.
3. The client lowers REQ_e to indicate that event e is finished.
4. The synchronizer lowers ACK_e indicating that clients may make another request for event e.

Therefore, to use the synchronizer, a client wishing to perform event *e* waits for ACK_e to be lowered, then raises REQ_e to request permission. When ACK_e is raised, the client proceeds with event *e*, then lowers REQ_e. This four-cycle protocol eliminates the need for a global clock.

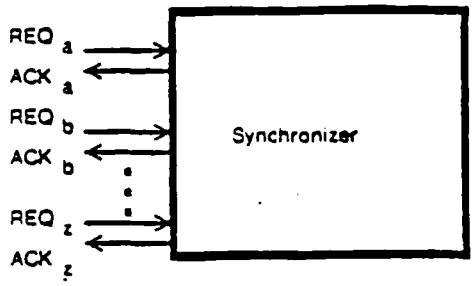

Figure 1: A synchronizer

In the description of a synchronizer to Miss Manners, events are declared in a statement of the form:

```
EVENT start, transmit, finish
```

This signals to Miss Manners that start, transmit, and finish are events, and that a request-acknowledge pair should be allocated for each one.

To describe the synchronization of events they can be combined into path expressions. We describe the syntax and give an intuitive definition of the semantics of a path expression. More rigorous definitions are given by several authors [1, 11]. A *simple path expression*, or simply *path*, is a regular expression with an outermost Kleene star. The operators that are permitted in the regular expression are the Kleene star (''*''), sequencing (indicated by juxtaposition), and union (''+'') operators. Operands in the expression are event names for the events to be synchronized. The outermost Kleene star is represented by the keywords `PATH . . . END`, which also delimit the simple path expression. For example, a* would be represented by `PATH a END`.

A path specifies the sequencing of events that appear within it in two ways. First, no two events appearing in a path may be concurrent. The time intervals during which the events occur may not overlap. The expression `PATH a + b END` thus specifies that events a and b may not occur at the same time. Second, the sequence of event occurrences for events that appear in a path must be a prefix of a string that is in the regular language specified by that path. Thus, the path `PATH a b END` indicates that occurrences of a must alternate with occurrences of b and that a and b cannot occur together.

To permit a more compact notation for some commonly occurring path expressions, we have added an additional construct called a *flag* that can be used in simple paths. Flags fulfill the function of state labels in other notations for regular languages [19], but seem to be more convenient within the context of path expressions. A flag can be thought of as an event that occurs inside the synchronizer; flags can be set and checked within a path. For example, the path:

```
EVENT store, fetch
FLAG f0^, f1, f2
PATH (f0: store f1^)
     + f1:(store f2^ + fetch f0^)
     + (f2: fetch f1^) END
```

represents an up-down counter that might be used to synchronize access to a bounded buffer of size 2. The statement:

```
FLAG f0^, f1, f2
```

indicates that `f0` is a flag that is initially set, while `f1` and `f2` are flags that are initially cleared. A construction such as `f0:` that appears in a sequence prevents events that follow it in the sequence from occurring unless the flag `f0` is set. If `f0` is set, it will be cleared by the `f0:` construction, and the tail of the sequence will be permitted. The construction `f0^` sets flag `f0`.

Flags permit transfer of control in the same way that state labels do. Setting a flag with the "^" operator corresponds to a go to, while checking a flag with ":" corresponds to a label. We prefer flags to state labels, however, because they permit more flexibility. One advantage is the ability to delay transfers of control. For example, if a subcircuit wants to alternately read and write from a bus that has a protocol for bus acquisition and freeing, the path for that subcircuit could look like this:

```
FLAG rlast^, wlast
PATH rlast: getbus write wlast^ freebus +
     wlast: getbus read rlast^ freebus END
```

In this path, the flags are set before the bus is freed. The ability to indicate a state change near the action that causes it rather than at the end of a long sequence can make paths more readable.

A *multiple path expression* is a set of simple paths. Each of the paths that make up a multiple path expression constrains the occurrences of events to a regular language. In addition, if an event appears in two separate paths, the event must satisfy each of the paths separately. Thus, in the multiple path expression:

```
PATH a b END
PATH c a END
```

event a cannot occur until c has occurred, and two occurrences of event a must be separated by precisely one occurrence of b and one of c.

We end the description of our path expression language with a complete example. This path expression enforces mutual exclusion of two subcircuits, A and B, on a bus. Subcircuit A has priority in that it can "reserve" the bus between transmissions and prevent subcircuit B from using it.

```
EVENT a, b
    /* A has the bus during event a */
    /* B during event b */

EVENT areserve, bcheck
    /* These are for priority */

PATH a + b END
    /* mutual exclusion */

PATH bcheck b END
    /* b must check before grabbing the bus */

PATH areserve + bcheck END
    /* if a is reserving, b can't check */
```

This path expression contains three simple paths. The first one prevents both A and B from using the bus at the same time. The second and third paths implement the priority scheme: B must check before it uses the bus, and A can prevent B's checking event from completing by reserving the bus.

Path expressions can describe many types of synchronization requirements. The path expression above describes one type of synchronization that might be used in a loosely-coupled system; other path expressions for different forms of synchronization have been presented by other researchers [3, 10]. The simplicity and expressiveness of path expressions make them an obvious candidate for describing synchronizers to a specialized silicon compiler.

3 Building Synchronizers

This section describes the overall structure of the synchronizers generated by Miss Manners. The synchronizers are based upon the design reported by Anantharaman *et al.* [1], but have been modified to reduce the number of circuit elements (Figure 2). Three subcircuits are shown in the diagram: the event handlers, the arbiter, and the sequencers. The event handlers are constructed from Muller C elements, inverters, and NOR gates, as shown in the diagram. The arbiter in Figure 2 performs mutual exclusion for events that occur in the same simple path. Signals to the arbiter are asserted low, so it ensures that at most one of the OUT signals for a mutually exclusive set of events is low, no matter how many of the IN signals are low. Finally, each of the sequencers shown in Figure 2 ensures that the sequence of events satisfies one of the simple paths; there is one sequencer for each simple path in a multiple path expression.

The interaction between the event handlers and sequencers is rather complex. 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. The TA_e lines from all sequencers for paths containing event *e* are fed into the Muller C gate for event *e*. The DIS_e lines from these sequencers are fed into the large AND gate for event *e*. Event handlers communicate with sequencers by means of a four cycle protocol on TR/TA, similar to the one on REQ/ACK. The TR_e signal tells a sequencer to update its state because of the occurrence of event *e*. The TA_e signal indicates that the sequencer is finished updating. When both TR and TA are low, the DIS lines indicate which events violate

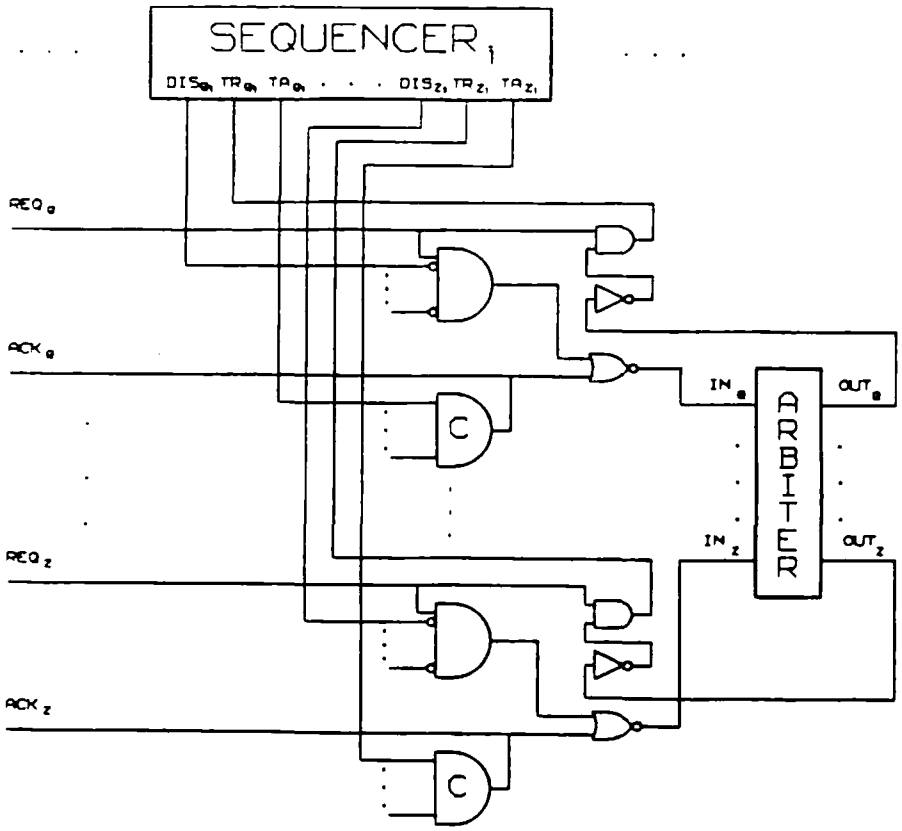

Figure 2: A synchronizer circuit

the simple path.

We now describe how the circuit works. A more complete description of a similar circuit, with a proof of correctness, is given elsewhere [1]. If a client raises REQ_e at a time when no sequencers are asserting DIS_e, IN_e is lowered. Eventually, the arbiter will decide in favor of IN_e, lowering OUT_e and raising TR_e. Each sequencer that has e in its path finds TR_e high, and some time later raises TA_e. When all sequencers have raised TA_e the output of the C gate goes high, raising ACK_e and latching IN_e low. When the client signals completion of e by lowering REQ_e, TR_e will be lowered. Each sequencer then finds TR_e low, and lowers TA_e; when all sequencers are done, ACK_e and IN_e are raised, clearing the arbiter and ending the event.

Two parts of the circuit in Figure 2 remain to be described: the sequencer and the arbiter. We begin with the sequencer, which contains a recognizer similar to the one described by Floyd and Ullman [7] together with a controller that generates the TA signals. A realization for the controller is shown in Figure 3. Along with the TA signals, it generates two timing signals for the recognizer, called **Start** and **End**. **Start** is true whenever at least one TR is on and no TA is on, while **End** is true whenever at least one TA is on and no TR is on. The realization in Figure 3 uses a delay element to generate the control signals, but a completely self-timed controller can be built if it is needed. The delay in Figure 3 must be long enough so that all of the actions started by TR can finish. An upper bound on the delay can be computed from the rest of the layout. The element labelled M. E. is a mutual exclusion element that guarantees that **Start** and

End are never high at the same time. Its construction is given by Seitz [17].

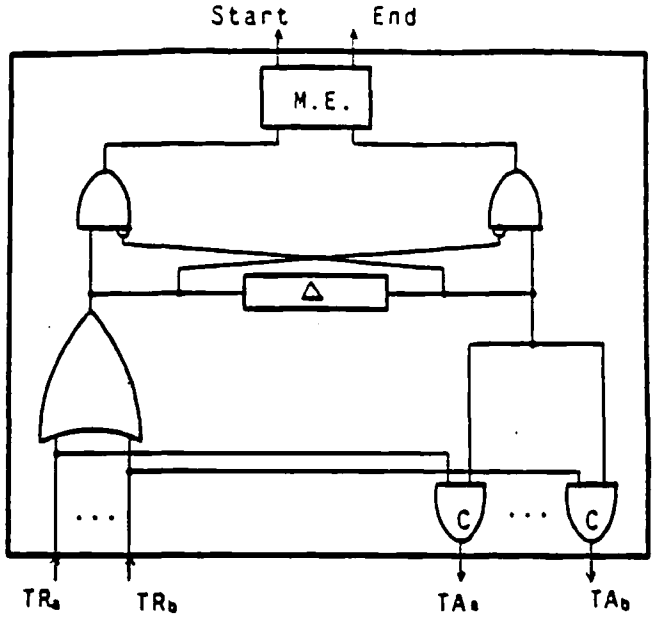

Figure 3: A controller for a simple path sequencer

A recognizer is made up of several types of cells, corresponding to the characters that can appear in a simple path. One type of cell, shown in Figure 4, corresponds to event names. The other types of cells correspond to operators, and to the setting and checking of flags. The ENB and RES signals in Figure 4 are used within the recognizer to keep track of which events in the path have occurred and which are legal. The ENB signal is true if and only if the event corresponding to the cell can occur next in the path. The RES signal is set to true after the event occurs.

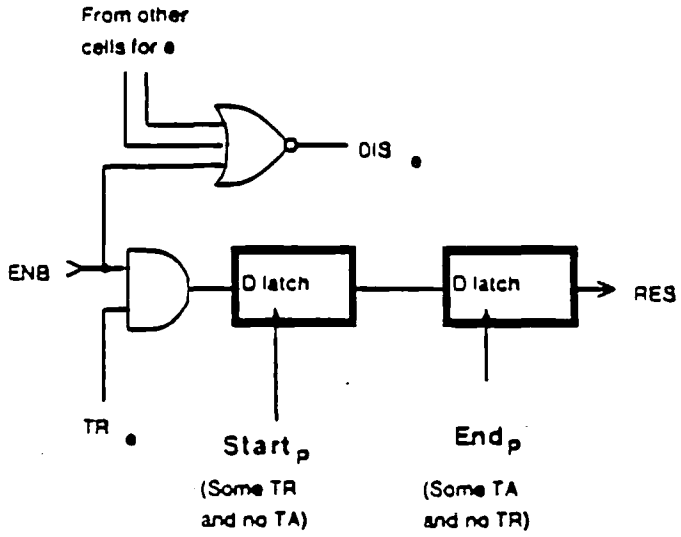

Figure 4: Cell for event *e*

The two latches in Figure 4 are clocked by the signals **Start** and **End**. They control the flow of ENB and RES signals so that the event cell propagates a 1 from ENB to RES only if event *e* occurs. When this cell is used in a recognizer for a simple path, the ENB input will be true if and only if event *e* is permitted by the path. Thus, if ENB is true it negates DIS 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 the 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 second latch when TR is lowered.

A recognizer is constructed from these cells by connecting the RES output of some of the cells to the ENB input of others. For example, a recognizer for the path `PATH a b c END` is simply a cascade of three event cells. The cells for the operators + and * route the RES and ENB signals from their operands so that the recognizer performs the correct function. Fuller details on these cells can be found in the references [1, 8, 9].

There are three types of cells related to flags. Two of these, shown in Figure 5, are the flag operator cells corresponding to the setting and checking of flags. The third cell, shown in Figure 6, contains the flag's state.

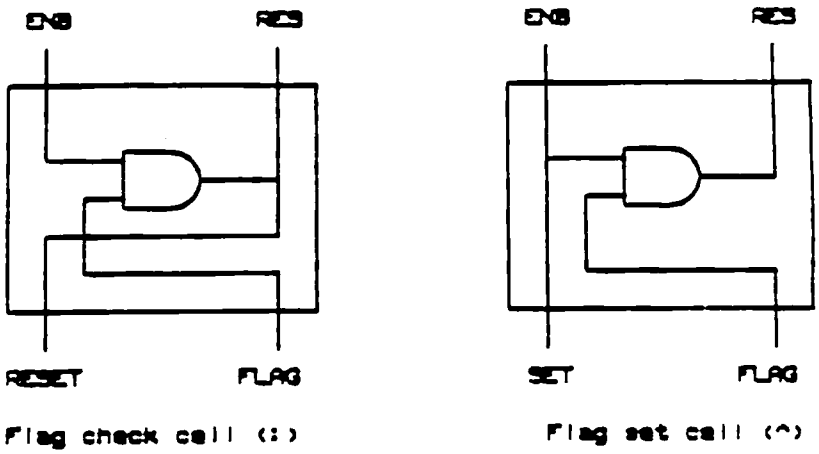

Figure 5: Flag operator cells

Like the other operator cells, the flag operator cells are simple combinatorial logic circuits that are connected to the rest of the sequencer by ENB and RES lines. However, they are also connected to the flag-state cells by SET, RESET and FLAG lines. The flag-set cell raises the SET line when its ENB line is raised. It then raises RES when its FLAG input goes high. The flag-check cell propagates a 1 from ENB to RES only if the flag is set; it also raises the RESET line.

The flag-state cell is a one-bit memory element which stores the status of the flag. The bit can be set asynchronously, but is reset only on the **End** signal following the raising of RESET. This ensures that any event enabled by the checking of the flag is completed before the flag is reset. The SET lines from the various flag-set cells corresponding to a flag are ORed to give the set signal for the memory element. The RESET line from the corresponding flag-check element is clocked by the **End** signal to give the reset signal for the memory element. The status of the flag is brought out on the FLAG line.

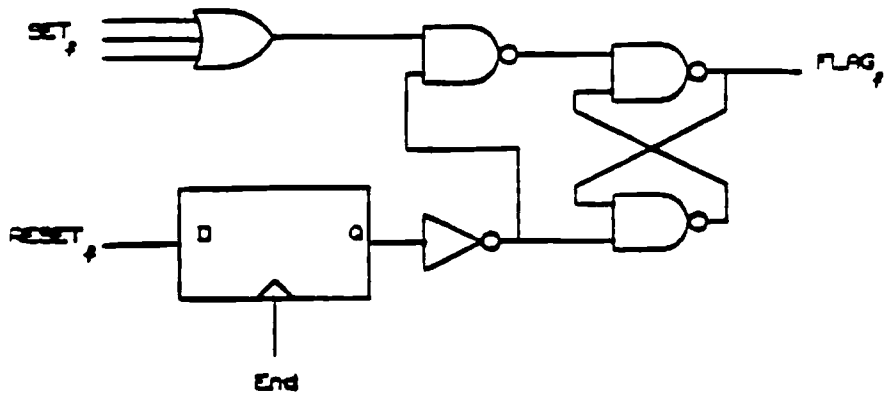

Figure 6: Cell for flag f

The sequencer is formed by combining a controller and recognizer as shown in Figure 7. Using tree layout techniques of Leiserson [12], the recognizer for an expression of length n requires only $O(n \log n)$ area.

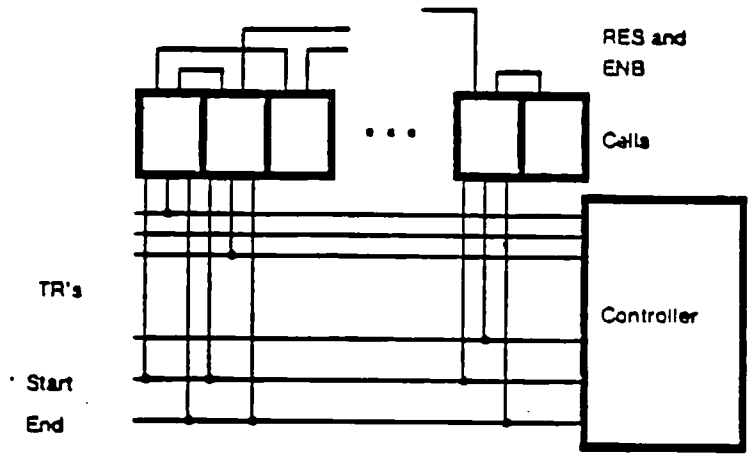

Figure 7: The floorplan for a sequencer

The other circuit from Figure 2 that must be implemented is the arbiter. This circuit selects a single event from a mutually exclusive set of requests. If the IN signals from a set of conflicting events are low, the arbiter ensures that only one of the OUT signals from that set is low. The arbiter should be fair if possible, so that any request that remains asserted will eventually be selected.

Since mutual exclusion is the only condition to be enforced, the arbitration function can be represented by a *conflict graph* in which each event corresponds to a vertex and an edge is drawn between two vertices if their events conflict. A single path corresponds to a conflict graph which is the complete graph on all of its events. The conflict graph for a multiple path expression is then just an overlapping set of these cliques. Figure 8 shows the conflict graph of a multiple path expression. The arbiter must select a set of vertices in the path expression that correspond to

requested events. No two vertices in the set selected may be connected by an edge.

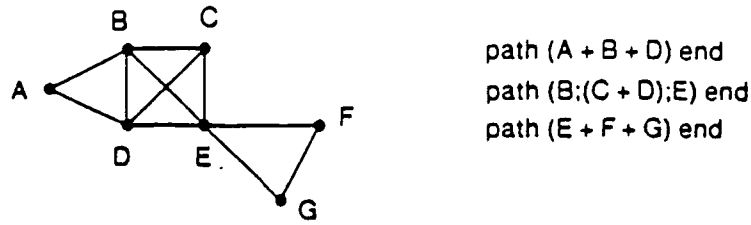

Figure 8: A path expression and its conflict graph

If the conflict graph is simple, so is the arbiter. For example, Seitz [16] has shown how to build an arbiter for the path expression **path A + B end.** This arbiter simply ensures that only one of A or B is enabled, and can be constructed using an interlock element as shown in Figure 9. This interlock has been fabricated and used in practical circuits [14].

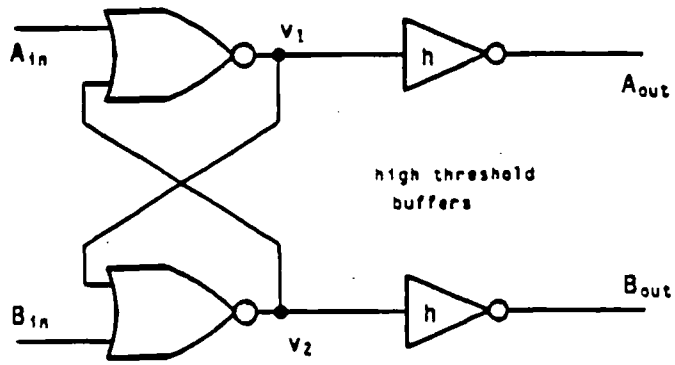

Figure 9: An interlock element

In this interlock, signals are asserted low so that if neither element is requesting then both inputs will be high. Suppose that this is the case, and that both v_1 and v_2 are near 0 volts. Then both outputs will be high. If A_{in} is lowered at this point then v_1 goes high and A_{out} goes low, signaling that A may proceed. Once this happens, B_{out} remains high for as long as A_{in} is kept low, so that B cannot occur until A is finished. Thus, the interlock enforces mutual exclusion as long as the inputs change one at a time.

If the inputs A_{in} and B_{in} change at nearly the same time, then the cross-coupled NOR gates enter a metastable state which persists for an unpredictable period of time. In this case the outputs of the NOR gates cannot be used directly as outputs of the interlock. The high-threshold buffers prevent false outputs during this metastable state.

The synchronizers produced by Miss Manners use an arbiter that extends Seitz's idea by generalizing it to any conflict graph. Each vertex in the graph corresponds to a NOR gate, and each edge corresponds to a cross-coupling of two NOR gates. An arbiter of this sort can be compactly laid out as a Weinberger array [20]. The NOR gates are laid out horizontally, with routing tracks in the array running vertically. Each track carries the output of a NOR gate to the inputs of the gates with which it is cross-coupled. In some cases a track may be able to carry

more than one output signal. The outputs of the array pass through Schmitt triggers whose trigger points are set well above the logic threshold of the NOR gates.

Miss Manners as currently implemented consists of three phases. The first takes the synchronizer description in the path expression language described earlier and produces a parse tree and block diagram. The second takes the parse tree and block diagram as input and produces a detailed floor-plan. The third produces an NMOS layout from the floor-plan.

Miss Manners is intended as a real design tool, which we hope will be widely used. If designers are to want to use it, Miss Manners must produce circuits that are not significantly larger or slower than those that can be produced by hand. A few simple modifications to the basic compilation scheme described above can reduce the size of circuits, while at the same time increasing their speed.

One simple optimization is the compaction of the arbiter Weinberger array. Since one NOR gate is needed for each event, the number of NOR gates in the array is fixed, but the number of tracks that are used may vary. Traditionally, the number of tracks in a Weinberger array is minimized by permuting the NOR gates [2]. Finding the smallest array is an NP-complete problem. Our problem, fortunately or not, is more restricted and hence simpler. Because Miss Manners produces parts of larger circuits, the spatial ordering of the REQ and ACK lines may affect the wiring outside of the synchronizer. Miss Manners therefore does not change this ordering. A designer can specify the ordering of these lines in the final layout: the order in which events are declared in EVENT statements is the top-to-bottom order of their interface wires. Since the order of events dictates the order of NOR gates in the arbiter, the compiler cannot permute the gates. Thus, the only layout optimization that can be done is to pack interconnections into as few tracks as possible. For example, Figure 10 shows a badly laid-out arbiter, while Figure 11 shows one that is well laid out. The horizontal lines in the figures represent NOR gates. The vertical lines represent tracks, with gate outputs represented by circles and inputs represented by boxes.

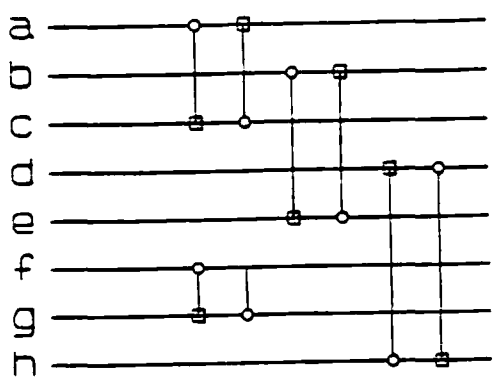

Figure 10: A badly laid-out arbiter

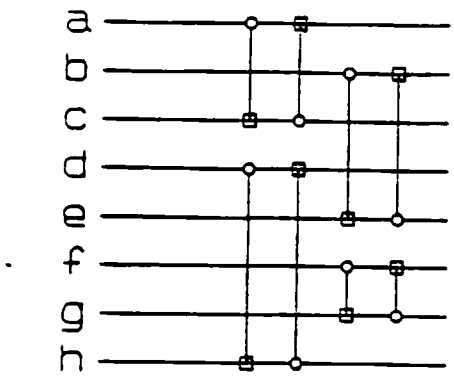

Figure 11: A well laid-out arbiter

The restricted layout problem, with no permutation of columns, can be solved optimally by a greedy algorithm. To lay out a set of edges, sort them so that their topmost endpoints are in descending order. All edges with an endpoint in the topmost channel should come first, followed by all edges with an endpoint in the channel second from the top, and so on. Remove edges from the sorted list in order and place each edge in the leftmost track in which it fits. This first-fit algorithm will use the minimal number of tracks to lay out the set of edges.

To prove that this greedy algorithm finds an optimal layout, let G be the greedy layout and consider an optimal layout L of the set of edges. Sort the edges as in the greedy algorithm, remove them from the list and order, and place them as in layout L. Let *e* be the first edge whose placement differs in G and L; suppose it is in track *i* in G and track *j* in L. Place *e* in track *i*, and continue performing the optimal layout with tracks *i* and *j* switched. Clearly this can be done, since:

- only tracks *i* and *j* are affected;
- any edge that L placed in track *i* can now be placed in track *j* since *j*'s empty space includes the empty space that *i* had;
- any edge that L put in track *j* can now be put in track *i* since *i* has exactly the same empty space that *j* did.

The process of moving edge *e* and switching tracks *i* and *j* thus produces a new optimal layout L_1 whose first difference from G is after edge *e*. Continuing this process with subsequent differing edges will eventually produce layout G, showing that G is optimal.

The arbiter is not the only structure in a synchronizer that needs to be optimized. Some of the sequencers may be superfluous, and these can be removed. Some paths, such as `PATH a + b END`, specify only mutual exclusion; no sequence is ruled out by such paths. The sequencer corresponding to paths such as this never disables an input, and so can be eliminated without affecting the function of the synchronizer.

This section has summarized a scheme for compiling a path expression into a layout and has

outlined a few of the optimizations that are performed by Miss Manners. These optimizations are possible because Miss Manners is a specialized silicon compiler; the specialized knowledge of synchronizers that is needed for optimal layout can be encoded into the program. Optimizations such as these are required in any compiler that is to produce layouts that are as efficient as those designed by hand.

4 Status and Plans

A first version of Miss Manners is running at Columbia. It accepts a path expression written in the language described in Section 2 and produces the floorplan of a synchronizer for that expression. The optimizations described in Section 3 are applied to produce a compact floorplan. Using standard cells, an NMOS layout for the entire synchronizer is produced.

We foresee several directions for future work on compilers of this type. Perhaps the most obvious is the design of primitive cells in CMOS so that Miss Manners can produce CMOS circuits. We also intend to incorporate other layout optimizations; for example, alternative strategies may produce more compact layouts of sequencers [18]. Another area where more work is needed is the design of arbiters. The cross-coupled NOR gates proposed here can starve clients under some circumstances [1] and may require too much time to resolve metastability. Experience will show whether these problems arise in practice; if they do, other arbiter designs will be tried.

Miss Manners is ready for use by designers of loosely-coupled systems; future developments will increase its usefulness. We encourage designers to make use of this tool. It produces efficient circuits from simple behavioral descriptions. Specialized silicon compilers of this type can ease the design of efficient circuits and help fulfill the promise of VLSI.

References

[1] Anantharaman, T. S., E. M. Clarke, M. J. Foster, and B. Mishra.
Compiling Path Expressions into VLSI Circuits.
In *Proceedings of the 12' th Symposium on Principles of Programming Languages*. ACM, January, 1985.
An expanded version of this paper will appear in *Distributed Computing*.

[2] Asano, T.
An Optimum Gate Placement Algorithm for MOS One-Dimensional Arrays.
Journal of Digital Systems VI:1-27, 1981.

[3] Campbell, R. H. and A. N. Habermann.
The Specification of Process Synchronization by Path Expressions.
In G. Goos and J. Hartmanis (editor), *Lecture Notes in Computer Science, Volume 16*, pages 89-102. Springer-Verlag, 1974.

[4] Chaney, T. J. and C. E. Molner.
Anomalous Behavior of Synchronizer and Arbiter Circuits.
IEEE Transactions on Computers :421-422, April, 1973.

[5] Chapiro, D. M.
Globally-Asynchronous Locally-Synchronous Systems.
PhD thesis, Stanford, October, 1984.

[6] Dijkstra, E. W.
Cooperating Sequential Processes.
In Genuys (editor), *Programming Languages*, pages 43-112. Academic Press, New York, 1968.

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

[8] Foster, M. J.
Specialized Silicon Compilers for Language Recognition.
PhD thesis, CMU, July, 1984.
This thesis will appear as a chapter in *VLSI Electronics: Microstructure Science*, Volume 14, edited by N. Einspruch and published by Academic Press.

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

[10] Kolstad, R. B. and R. H. Campbell.
Path Pascal User Manual
Department of Computer Science, University of Illinois, Urbana, Ill. 61801, 1980.

[11] Lauer, P. E. and Campbell, R. H.
Formal Semantics of a Class of High-Level Primitives for Coordinating Concurrent Processes.
Acta Informatica 5:297-332, June 5, 1974.

[12] Leiserson, C.E.
Area-Efficient VLSI Computation.
PhD thesis, Carnegie-Mellon University, 1981.

[13] Li, W. and P. E. Lauer.
A VLSI Implementation of Cosy.
Technical Report ASM/121, Computing Laboratory, The University of Newcastle Upon Tyne, January, 1984.

[14] Lyon, R. F. and M. P. Haeberli.
Designing and Testing the Optical Mouse.
VLSI Design III(1):20-30, January/February, 1982.

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

[16] Seitz, C. L.
Ideas About Arbiters.
LAMBDA First Quarter:10-14, 1980.

[17] Seitz, C.
System Timing.
Chapter 7 of "Introduction to VLSI Systems" by Mead and Conway, Addison Wesley, 1982.

[18] Trickey, H. W.
Good Layouts for Pattern Recognizers.
IEEE Transactions on Computers C-31(6):514-520, June, 1982.

[19] Ullman, J. D.
Combining State Machines and Regular Expressions for Automatic Synthesis of VLSI Circuits.
Technical Report CS-82-927, Stanford University, September, 1982.

[20] Weinberger, A.
Large Scale Integration of MOS Complex Logic: A Layout Method.
IEEE Journal of Solid State Circuits SC 2, February, 1967.