Optimizing Synchronous Systems

Charles E. Leiserson

Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

James B. Saxe

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

Abstract—The complexity of integrated-circuit chips produced today makes it feasible to build inexpensive, special-purpose subsystems that rapidly solve sophisticated problems on behalf of a general-purpose host computer. This paper contributes to the design methodology of efficient VLSI algorithms. We present a transformation that converts synchronous systems into more time-efficient, *systolic* implementations by removing combinational rippling.

The problem of determining the optimized system can be reduced to the graph-theoretic *single-destination-shortest-paths* problem. More importantly from an engineering standpoint, however, the kinds of rippling that can be removed from a circuit at essentially no cost can be easily characterized. For example, if the only global communication in a system is *broadcasting* from the host computer, the broadcast can always be replaced by local communication.

1. Introduction

Gone are the days when the chips in an electronic system were mostly NAND gates and flip-flops. The computer-on-a-chip is the fashion today, and microprocessors are being produced with ever-increasing functionality in an attempt to exploit the improving technological capability. In the future, we can expect designers to put entire multiprocessor systems on a single chip. By exploiting the massive potential that VLSI holds for parallel computation, such special-purpose systems will greatly augment the power of general-purpose computing environments.

Designers of these large systems will face the problem of *communication*, which arises in any parallel system. In order to cope with communication costs, the design methodology of *systolic systems* [5, 6] has been proposed. This design methodology allows a high-level, algorithmic description of a circuit which deals directly with communication costs. As a result, the performance of circuits implemented with this design methodology matches well the performance predicted by a simple algorithmic analysis [3]. Signal processing, numerical linear algebra, and raster graphics provide many applications for systolic systems, but their utility is hardly restricted to these areas alone.

The systolic design methodology manages communication costs effectively because *the only communication permitted during a clock cycle is between a processing element and one of its neighbors* in the communication graph of the system. This constraint is in direct contrast with, for example, the propagation of a carry signal which *ripples* down the length of an adder. Such combinational rippling and global control such as *broadcasting* are forbidden in systolic designs. Whereas the clock period in most synchronous systems is long in order to allow signals to ripple through cascades of combinational logic, in systolic systems the clock period depends only on the time required for a signal to propagate through a single processing element. *Thus the clock period of a systolic system is short because it is independent of the size of the system.*

The principal deficiency of the systolic design methodology is that the burden on the designer may be excessive. In particular, global communication such as *broadcasting* is more easily described in terms of rippling logic. In a systolic system the effect of broadcasting must be achieved by multiple local communications. The propagation of a datum from one end of a systolic system to the other may take many ticks of the system clock, and therefore different processing elements will see the same data at different times. Orchestrating the individual behaviors of processors can be a large bookkeeping chore.

In this paper, we address the design issue head on. We demonstrate how a synchronous system can be designed with rippling logic, and then converted to a systolic implementation which is functionally equivalent to the original system—the principal difference being the shorter clock period of the systolic implementation. We characterize the systems that can be so transformed, and show that an algorithm based on the graph theoretic *single-destination-shortest-paths* problem can compute the transformation quickly.

The remainder of this paper is organized as follows. Section 2 presents a graph-theoretic model for synchronous systems. The principal result of the paper is given as the *Systolic Conversion Theorem* in Section 3. To illustrate the power of the theorem in a design situation, Section 4 details the construction of a simple systolic system. The design process from Section 4 is abstracted in Section 5 and applied

This research was supported in part by the Defense Advanced Research Projects Agency under Contract No. N00014-80-C-0622 and by the Office of Naval Research under Contract N00014-76-C-0370.

CH1695-6/81/0000/0023$00.75 © 1981 IEEE

to the problem of replacing global communication in a system with local communication. Section 6 is devoted to brief discussions of further topics relating to applications and extensions of the results. Section 7 presents some concluding remarks.

2. A Model for Synchronous Systems

We view a large circuit as a system which is partitioned into *functional elements* (combinational logic) and *registers* (clocked memory). Such a system S can be modeled as a finite *edge-weighted directed multigraph* (henceforth, we shall simply say "graph") $G = (V, E)$. The *vertices* V of this *communication graph* correspond to functional elements, and the *edges* E correspond to interconnections between the functional elements. Each edge e in E is a triple of the form (u, v, w), where u and v are (possibly identical) vertices of G, called the *tail* and *head* of e, and w is the nonnegative integer *weight* of the edge. The weight is the number of registers along the interconnection between the two functional elements. A *configuration* of a system is some assignment of values to all its registers. With each clock tick, the system maps the current configuration into a new configuration.

If the weight of an edge happens to be zero, no register impedes the propagation of a signal along the corresponding interconnection. Such is the case, for instance, in the system S_1 shown in Figure 1.. A signal

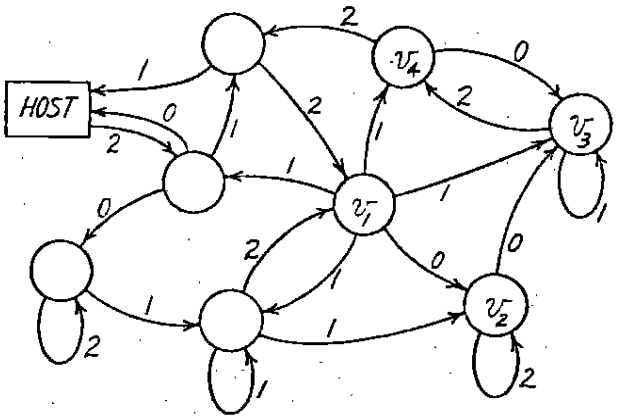


Figure 1: The communication graph G_1 of a synchronous system S_1. One vertex of the graph is distinguished as representing not an ordinary functional element but a *host processor*, which is the system's only interface to the external world.

emanating from v_1 will proceed unhindered though v_2 and subsequently through v_3 before it is stopped by a register. If the rippling can feed back on itself, problems of latching, oscillation, and race conditions can arise. In our model this corresponds to a zero-weight cycle in the communication graph, where the weight of a path is taken to be the sum of the weights of its edges. In order to preclude anomalous behavior associated with asynchronous design, we restrict our attention to *synchronous* systems.

> **Definition:** A system is a *synchronous system* if every cycle in its communication graph has positive weight.

Systolic systems exhibit no combinational rippling, and in our model correspond to those synchronous systems whose edge weights are all positive.

> **Definition:** A synchronous system S is a *systolic system* if for each edge (u, v, w) in the communication graph of S, the weight w is strictly greater than zero.

So far we have dealt with the internal organization of systems, but of course, no system operates in a vacuum. Rather, it communicates with the external world via an interface. In our model, one vertex called the *host* represents this external interface. If the system were an integrated circuit, for example, the connections to the host might be the pins of the chip.

The host is important because when two systems are compared, it is the behavior of the systems from *the point of view of the host* that is at issue.

> **Definition:** Let c be a configuration of a synchronous system S, and let c' be a configuration of another synchronous system S'. The system S started in configuration c has the same *behavior* as the system S' started in configuration c' if for any sequence of inputs to the system from the host, the two systems produce the same sequence of outputs to the host.

Before going on, we briefly call attention to a technical fine point relating to this definition. While the combinational logic in a system is settling, the system outputs to the host may change. We insist, however, that the internal state of the host depend only on the final settled values of the system outputs and not on any transient values. Furthermore, if signals ripple through the host to the system inputs during a single clock cycle, we similarly require that the eventual values of the system inputs not depend on any transient values. Of course in a synchronous system it is impossible for transient values on the system inputs to ripple back out to the host.

Suppose the system S_1 from Figure 1 is modified so that the weight of every edge leading into v_3 is increased by one and the weight of every edge leading out of v_3 is decreased by one (see Figure 2). As viewed from the host, the resulting system S_2 is equivalent (in a sense that we will presently make precise) to S_1. The two systems differ internally in that each computation performed by the functional element at v_3 in S_2 is performed one clock tick later than the corresponding computation in the original system S_1. Vertex v_3 is said to *lag* by one tick in S_2 with respect to S_1, or alternatively, to *lead* by one tick in S_1 with respect to S_2.

This simple operation of *retiming* a functional element in a system forms the basis of the optimization techniques in this paper. Indeed, the system S_2 exhibits a performance improvement over S_1 because the

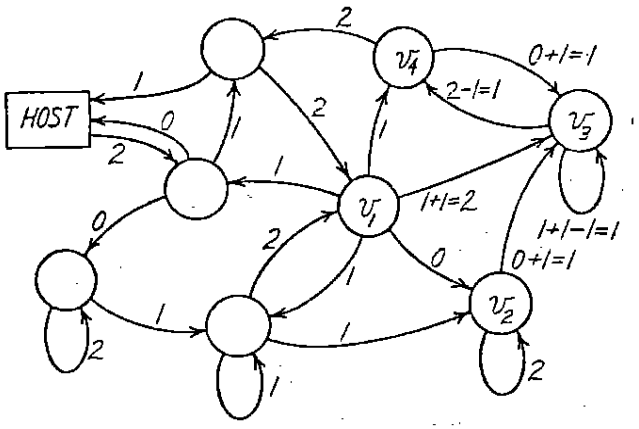


Figure 2: The communication graph of a system S_2 which is equivalent to the system S_1 from Figure 1 as viewed from the host. Internally, the two systems differ in that vertex v_3 lags by one clock tick in S_2 with respect to S_1.

longest path of combinational rippling has been shortened by the retiming of the functional element at v_3. But although it may appear intuitive that the two systems are effectively the same from the point of view of the host, a closer examination reveals some technical difficulties. In particular, it is not true that whatever configuration the system S_1 is started in, there exists a configuration in which the new system S_2 can be started such that the two systems exhibit the same behavior. If the first register on the edge from v_3 to v_4 is initialized in S_1 with a value that is not in the range of values produced by the functional element at v_3, then the transformed system S_2 cannot necessarily mimic the subsequent behavior of S_1.

Although the system S_2 cannot mimic the behavior of S_1 for *all* possible configurations of S_1, however, it can for any sufficiently old configuration of S_1, that is, any that arise after S_1 has been run sufficiently long. In the example, a configuration of S_1 that has at least one predecessor is sufficiently old to allow S_2 to mimic the subsequent behavior.

> **Definition:** Let S and S′ be synchronous systems. Suppose that for every sufficiently old configuration c of S, there exists a configuration c' of S′ such that when S is started in configuration c and S′ is started in configuration c', the two systems exhibit the same behavior. Then system S′ can *simulate* S. If two synchronous systems can simulate each other, they are *equivalent*.

The following lemma shows that retiming a vertex in a synchronous system, as in the example above, produces an equivalent system as long as no edges in the communication graph are given negative weight by the transformation. In fact, the *Retiming Lemma* shows that many functional elements in a system can be given arbitrary leads and lags under the same condition.

> **Lemma 1:** (*Retiming Lemma*) Let S be a synchronous system with communication graph G, and let *lag* be a function that maps each vertex v of G to an integer and the host to zero. Suppose that for every edge (u, v, w) in G the value $w+lag(v)-lag(u)$ is nonnegative. Let S′ be the system obtained by replacing every edge $e = (u, v, w)$ in S with $e' = (u, v, w+lag(v)-lag(u))$. Then the systems S and S′ are equivalent.

Sketch of proof. The proof is an induction argument. Let t_0 be the maximum lag of any vertex in G. If S is started at time zero and run until time t_0, there exists a configuration of S′ such that for all t greater than or equal to t_0, each functional element v in S′ performs the same computation at time t as the corresponding functional element in S at time $t-lag(v)$. Appendix A gives a detailed proof of this lemma. □

Two synchronous systems may be equivalent even though their internal organizations are radically different. For example, one system might be a tree and the other a mesh. A system S′ obtained by retiming a system S, however, is not only equivalent, but also has the same *structure* as S.

> **Definition:** Two systems have the same *structure* if they are composed of the same functional elements with the same inputs and outputs connected by the same interconnections. The numbers of registers on the interconnections, however, may differ.

Preservation of structure is a valuable property of any system transformation because it allows the new system to inherit the benefits of independent design decisions. For example:

- Anything done to make the functional elements in the original system small or fast will carry through to the new system.
- Any topological properties of the communication graph that lead to an area-efficient layout of the original system will retained by the new system. If the original system is structured as a mesh, for instance, the new system will not be converted to, say, a shuffle-exchange graph, which is much more expensive to lay out [13].
- A system that has been partitioned across multiple chips will not require additional pins for the transformed system.

3. The Systolic Conversion Theorem

The previous section demonstrated that a synchronous system could be modified by *retiming* functional elements to reduce the period of the system clock. A natural question to ask is, "Is there a way to assign a lag to each vertex of the communication graph G of a synchronous system so that the retimed system is systolic?" The answer depends upon the *constraint graph* $G-1$, which is the graph obtained from G by replacing every edge (u, v, w) with $(u, v, w-1)$.

Theorem 2: (*Systolic Conversion Theorem*) Let S be a synchronous system with communication graph G, and suppose the constraint graph $G-1$ has no cycles of negative weight. Then there exists an equivalent system S′ which has the same structure as S and which is systolic.

Proof: The desired system S′ may be constructed by a procedure whose key step is the solution of a *single-destination-shortest-paths problem* in $G-1$. Without loss of generality, suppose that there exists a path from every vertex v in G to the host—whenever there is not such a path, it is impossible for the functional element at v to have any influence on the behavior of S. A corresponding path must exist in $G-1$, and since $G-1$ is finite and has no negative cycles, there must exist in $G-1$ a path of minimal weight from each vertex to the host. For each vertex v define $lag(v)$ as the weight of any such *shortest path* from v to the host in $G-1$. The systolic system S′ is obtained by modifying S using the Retiming Lemma (Lemma 1) to give each vertex the designated lag.

To show that this construction indeed produces a systolic system, we must demonstrate that all edge weights in the communication graph G' of S′ are strictly positive. This demonstration will also show that the conditions of the Retiming Lemma are met because all edge weights in G' will perforce be nonnegative. For any edge (u, v, w) in G, the weight of the corresponding edge in $G-1$ is $w-1$, and the weight of a shortest path in $G-1$ from v to the host is $lag(v)$. The weight $lag(u)$ of a shortest path in $G-1$ from vertex u to the host can be no greater than $(w-1)+lag(v)$. (Consider the path obtained by prepending the edge $(u, v, w-1)$ to a shortest path from vertex v to the host.) The weight of the edge $(u, v, w+lag(v)-lag(u))$ in G' is therefore positive. □

The Systolic Conversion Theorem can be applied to the system S_1 from Figure 1. Figure 3 shows the constraint graph G_1-1 for that system with the weights of the shortest paths to the host labeled in the vertices. Using the Retiming Lemma to give each vertex in S_1 the lag designated by the weight of its shortest path to the host yields the systolic system S_3 which is shown in Figure 4.

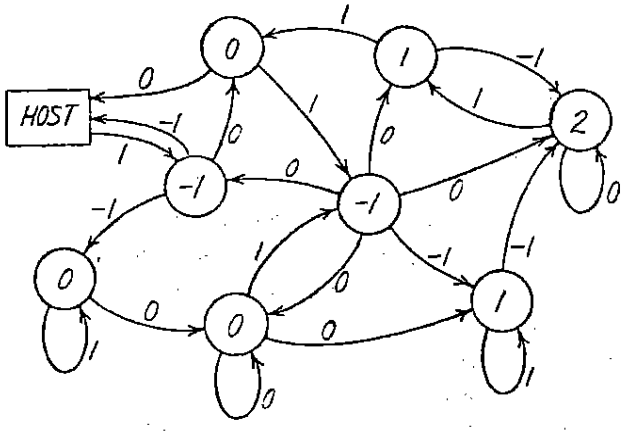


Figure 3: The *constraint graph* G_1-1 for the system S_1 in Figure 1. This graph is identical to G_1 except that the weight of each edge in G_1-1 is one less than the weight of the corresponding edge in G_1. Each vertex has been labeled with the weight of the shortest (*i.e.*, lightest) path from that vertex to the host.

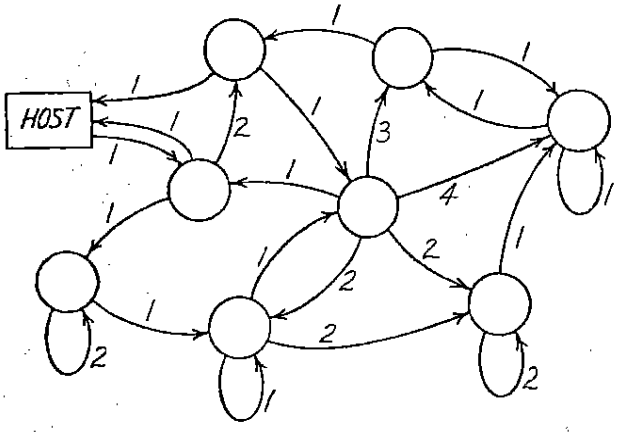


Figure 4: The systolic system S_3 produced by applying the Systolic Conversion Theorem to the system S_1 from Figure 1. Each edge (u, v, w) from G_1 has been replaced by the edge $(u, v, w+lag(v)-lag(u))$, where the lag of a vertex is taken as the weight of the shortest path from that vertex to the host in G_1-1 as indicated in Figure 3.

The Systolic Conversion Theorem shows that the absence of negative weight cycles in the constraint graph $G-1$ allows a synchronous system to be transformed by application of the Retiming Lemma into an equivalent systolic system. We call the graph $G-1$ a "constraint graph" because for each edge (u, v, w) of G, the edge $(u, v, w-1)$ in $G-1$ constrains the weights $lag(u)$ and $lag(v)$ of the shortest paths from u and v to the host to satisfy the inequality $lag(u) \leq lag(v)+w-1$, thereby guaranteeing that the edge $(u, v, w+lag(v)-lag(u))$ in G' will have positive weight.

The applicability of the Systolic Conversion Theorem to a system S does not depend on the specific functions computed by the functional elements of S. The next theorem shows that the Systolic Conversion Theorem is the strongest possible result of such generality. (Figure 5 shows the situation described in the statement of the theorem.)

Theorem 3: Let G be the communication graph of a synchronous system S, and suppose the constraint graph $G-1$ has a cycle c of negative weight and a path p from some vertex v on c to the host. Then it is impossible to construct a systolic system that both simulates S and has the same structure as S without using knowledge of the particular functional elements of S.

Proof: Suppose there does exist a structure-preserving transformation that produces a systolic system S′ which is equivalent to S. An adversary argument shows that this transformation must use specific properties of the functional elements of S. The adversary chooses functional elements for a system T which has the same structure as S, and then the transformation that produced S′ from S is applied to T. The resulting systolic system T′ fails to simulate T.

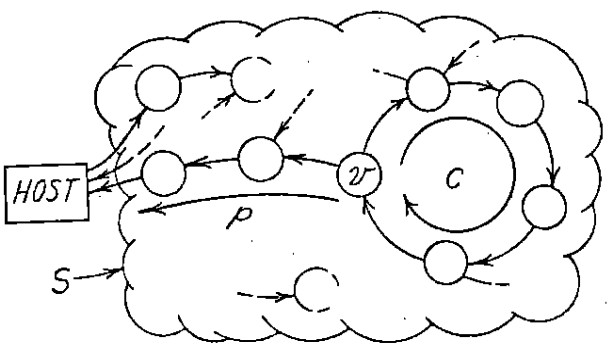


Figure 5: A cycle c and a path p from a vertex v of c to the host in the communication graph of a synchronous system.

Let G' be the communication graph of S', and let x and x' be the weights of cycle c in G and G'. Since system S' is systolic, the weight x' must equal or exceed the number of edges along c. Cycle c has negative weight in $G-1$, however, and thus x must be strictly less than the number of edges along c. Therefore, x is strictly less than x'.

The adversary now chooses the functional elements of the system T, which has the same communication graph G as S. Functional elements that lie neither on the cycle c nor on the path p from v to the host are chosen arbitrarily. Those on c and p pass their input values along c or p unchanged, except for the single functional element at v which takes an integer input a along c and propagates $(a+1) \bmod xx'$ along c and p. After a sufficiently long time, the outputs from T to the host along p at times t and $t+xx'$ will always differ by x' (modulo xx'). Now consider the system T' which is produced by applying to T the same transformation that produced S' from S. Since T' has x' registers along the cycle c', the outputs from T' to the host along p' at times t and $t+xx'$ will differ by x (modulo xx'). Since x and x' are distinct, the system T' cannot possibly simulate T. □

When a synchronous system with regular structure—such as a rectangular mesh or a complete binary tree—is converted to a systolic system, the conversion will typically be very uniform. The lead or lag of a functional element may, for example, be equal to the sum of its indices in the mesh or to its depth in the tree. Section 4 provides an example in which the functional elements are arranged in a linear array, and the lead of each functional element in the transformed system turns out to be its index in the array. For systems with less regular structure, however, an algorithm may be needed to determine the transformed system, if indeed the original system can be transformed.

The central computational problem involved is the *single-destination-shortest-paths problem*: Given an edge-weighted directed graph $G = (V, E)$ and a distinguished vertex d in V, find for each vertex v in V the weight of the shortest path from v to d, or else detect that G contains a cycle of negative weight. This problem can be solved for arbitrary graphs in $O(|V| \cdot |E|)$ time in the worst case by an algorithm due to Bellman and Ford [7, pp. 74–75]. Lipton, Rose, and Tarjan [10] have shown that this bound can be improved for families of graphs that have *small separators* using a method they call *generalized nested dissection*. The use of separators is particularly interesting because graphs with small separators have area-efficient layouts as well [8]. For any family of graphs that is closed under subgraphs and that has an n^α-separator theorem, where n is the number of vertices and $\alpha > 1/3$, the method of generalized nested dissection can be used to solve the single-destination-shortest-paths problem in $O(n^{3\alpha})$ time, provided that separators can be computed quickly enough. Thus a system with a planar communication graph can be transformed in $O(n^{3/2})$ time because planar graphs have a $\sqrt{n}$-separator theorem. Also, there is a $O(A^{3/2})$ time algorithm for any graph that has a VLSI layout with area A. (Use a homeomorphism between such graphs and subgraphs of cubic meshes of fixed depth, which have a $\sqrt{A}$-separator theorem.) Further improvements over the Bellman-Ford algorithm can be obtained for families of graphs with smaller separators.

4. A Real-Time Palindrome Recognizer

This section illustrates the power of the Systolic Conversion Theorem through an example. Although many more applications of this theorem may be found in such areas as signal processing or numerical linear algebra, we have chosen a simple symbol manipulation problem. A multitude of other applications of the theorem can be found in [9].

A string of n characters is a *palindrome* if the ith character for $i = 1, \ldots, n$ is the same as the $n-i+1$st character. Cole [2] constructs a linearly connected systolic array which is supplied characters from a string, and for each character tells immediately whether the string input up to that character is a palindrome. Whereas Cole constructs this real-time palindrome recognizer explicitly, his construction is elaborate and unintuitive. Here we demonstrate the same result, but the Systolic Conversion Theorem greatly simplifies the construction.

The construction of the palindrome recognizer is based on a linearly connected systolic array of "processors" which is augmented with rippling combinational logic. The Systolic Conversion Theorem is applied to this intermediate synchronous system to remove the rippling logic and yield once again a systolic array.

A processor p_i in the systolic array has two registers, A_i and B_i, each of which can hold either an ordinary character or a special *null character* denoted NIL. The contents of register A_i are provided as input to p_{i+1}. The host appears as p_0 in the system, and provides the input character in its register A_0.

The control of the processors is quite simple, but let us first understand what the systolic array is trying to do. Suppose the character string input to the systolic array is $x_1 \ldots x_n$. Figure 6 shows the contents of the A and B registers after ten characters have been input, and again after eleven. For $1 \le i \le \lfloor n/2 \rfloor$, register B in processor p_i contains the character x_i, and for $1 \le i \le \lceil n/2 \rceil$, register A in

i	1	2	3	4	5	6	7	8
A_i	x_{10}	x_9	x_8	x_7	x_6	NIL	NIL	NIL
B_i	x_1	x_2	x_3	x_4	x_5	NIL	NIL	NIL

i	1	2	3	4	5	6	7	8
A_i	x_{11}	x_{10}	x_9	x_8	x_7	x_6	NIL	NIL
B_i	x_1	x_2	x_3	x_4	x_5	NIL	NIL	NIL

Figure 6: Contents of the A and B registers.

processor p_i contains the character x_{n-i+1}. All other registers in the system contain NIL. Thus the input string is a palindrome if and only if whenever A_i and B_i are nonnull, they contain the same character.

On each clock tick, the new values of all the A_i and B_i are computed by the formulae

$$A_i \leftarrow \textit{if } B_{i-1} = \text{NIL } \textit{then } \text{NIL } \textit{else } A_{i-1}$$

and

$$B_i \leftarrow \textit{if } (A_i = \text{NIL}) \vee (B_i = \text{NIL}) \textit{ then } A_i \textit{ else } B_i,$$

where all assignments are performed simultaneously, so that all references to registers on the right hand sides denote values from the previous time step. While the host does not actually contain a register B_0, the system acts as if there were such a register which always had a nonnull value. Thus, A_1 is always given the value that A_0 held on the previous time step.

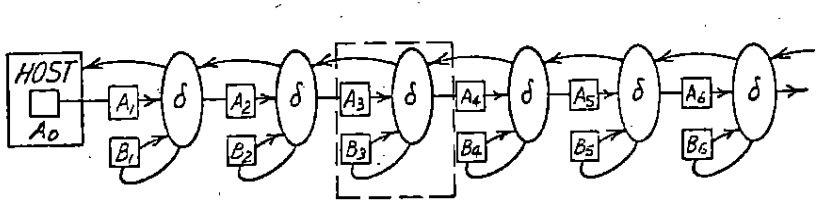
This system is now augmented by combinational logic which runs back to the host and reports whether a palindrome is recognized. This rippling "collection" logic returns TRUE if, for each processor p_i such that A_i and B_i both contain valid characters, $A_i = B_i$. The collection logic is implemented by giving each processor p_i an output PAL_i whose value is defined by the formula

$$PAL_i \triangleq (B_i = \text{NIL}) \vee (PAL_{i+1} \wedge (A_i = B_i)).$$

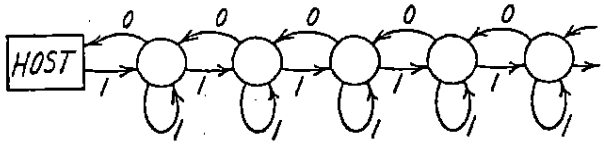
Since the PAL_i are propagated by rippling logic, the values on the right hand side of this definition are based on the current time step. The last processor in the array will have no PAL signal coming into it, but will act as if it had such a signal which always had the value TRUE.

The preceding text describes the palindrome recognizer in terms of "processors". Figure 7 shows how the recognizer may be modeled as a synchronous system P of functional elements and registers. Also shown in the figure is P's communication graph, which will be referred to as G for the remainder of this section. Figure 8 shows the state of P as it recognizes the palindrome "redivider".

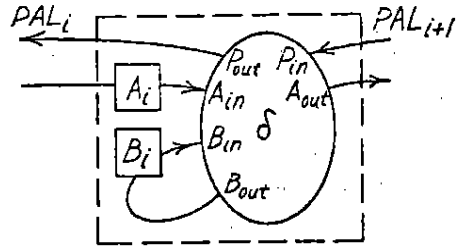
The chain of rippling logic that extends through all the functional elements to the host allows the recognition of a palindrome to be signalled on PAL_1 on the very next clock tick after the last letter of the



(a)



(b)



Specification of δ:

$$B_{out} \triangleq \begin{cases} \text{NIL} & B_{in} = \text{NIL} \\ A_{in} & B_{in} = \text{NIL} \end{cases}$$

$$A_{out} \triangleq \begin{cases} A_{in} & A_{in} = \text{NIL} \vee B_{in} = \text{NIL} \\ B_{in} & A_{in} \neq \text{NIL} \wedge B_{in} \neq \text{NIL} \end{cases}$$

$$P_{out} \triangleq (B_{in} = \text{NIL}) \vee (P_{in} \wedge (A_{in} = B_{in}))$$

(c)

Figure 7: A synchronous system P which recognizes palindromes in real time. (a) A diagram of P showing registers and functional elements. (b) The communication graph G for P. (c) Detail of a functional element.

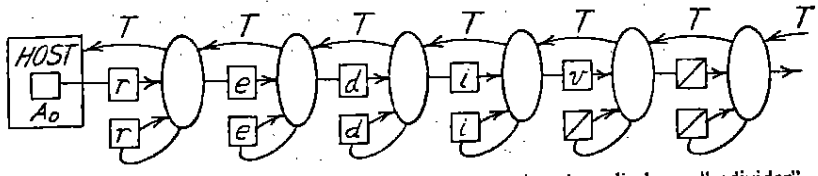


Figure 8: The configuration of the system P as it recognizes the palindrome "redivider".

palindrome becomes available in A_0. The disadvantage of this chain of rippling logic is that it must be allowed to settle after each clock tick, so the clock period of P must be linear in the length of the array. A systolic real-time palindrome recognizer with the same structure would be a great improvement since the clock period would be independent of the size of the system.

If we could verify that $G-1$ had no cycles of negative weight, we could use the Systolic Conversion Theorem to construct a systolic palindrome recognizer with the same structure as P. Unfortunately, $G-1$ has many cycles of negative weight (see Figure 9).

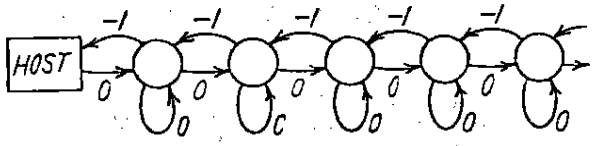


Figure 9: The constraint graph $G-1$ for the real-time palindrome recognizer P from Figure 7. Since this graph contains cycles of negative weight, the Systolic Conversion theorem is not applicable to P.

Consider, however, the system formed by modifying P so that the number of registers on each interconnection is doubled. The communication graph $2G$ of this system 2P is shown in Figure 10. All the data

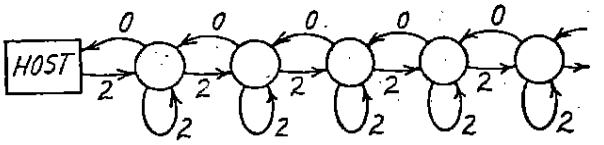


Figure 10: The communication graph $2G$ for a 2-slow simulator 2P of the system P from Figure 7. This graph is obtained by doubling the weight of every edge in the communication graph G of P.

flow in P is slowed down by a factor of two in 2P, so that 2P provides a sort of half-speed version of P which communicates with the host only on every other clock tick. In fact, 2P can be thought of as a pair of *2-slow simulators* of P—one communicating with the host on even-numbered ticks and the other processing a completely independent data stream and communicating with the host on odd-numbered ticks.

Unlike $G-1$, the constraint graph $2G-1$ is free of negative cycles (see Figure 11). Thus the Systolic Conversion Theorem can be applied

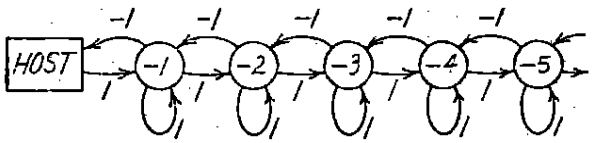


Figure 11: The constraint graph $2G-1$. Each vertex v is labeled with the weight of the shortest path from v to the host.

to 2P to produce a systolic array SP (shown in Figure 12) which recognizes palindromes in real time—two clock ticks per character.

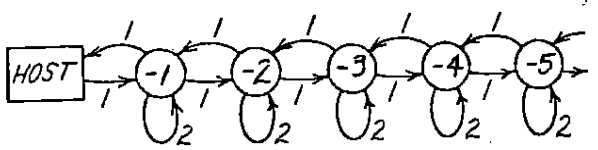


Figure 12: The communication graph of a systolic 2-slow simulator SP of the real-time palindrome recognizer P of Figure 7. Each vertex is labeled with its lag in SP with respect to the system 2P shown in Figure 10.

Each processor p_i has a lead of i in SP with respect to 2P. Figure 13 shows a sequence of internal states of SP as it recognizes the palindrome "redivider".

Although SP is a "2-slow" simulator of P, its performance is actually better than that of P if sufficiently large systems are compared. Since rippling in P runs the length of the system, the period of its clock must be at least proportional to the length of the array. System SP, which is systolic, can be run with a clock period that is constant with respect to the length of the system. Thus the duration of two clock ticks of SP will be less than the duration of one clock tick of P if the two systems are large enough.

As a comparison of Figures 8 and 13 reveals, system SP is much more complex than P in its internal workings. The recognition of any string as a palindrome by SP is spread out over several time steps, rather than happening all in a single clock cycle as in P. Consequently, any direct verification of SP's correctness requires careful bookkeeping to verify that all the data arrive at the right places at the right times. The effort involved in constructing such an argument is not superhuman—indeed, the correctness proof for Cole's palindrome recognizer [2] depends on just such a careful bookkeeping argument. It is the authors' contention, however, that it was an easier and less error-prone task to design and verify P than it would have been to design SP directly. The comparison would be even more favorable for a complicated system. In proving the Retiming Lemma and the Systolic Conversion Theorem, we have gone through the painstaking bookkeeping once and for all, and captured the result in a form that can be used again and again.

We can illustrate this point again by taking note of an interesting property of SP: supplying a NIL from the host to either of the two (odd ticks and even ticks) data streams being processed by SP effectively reinitializes the computation on that data stream. The reader may attempt to verify this property by a direct examination of SP, but we think it is easier to check that a NIL input effectively resets P.

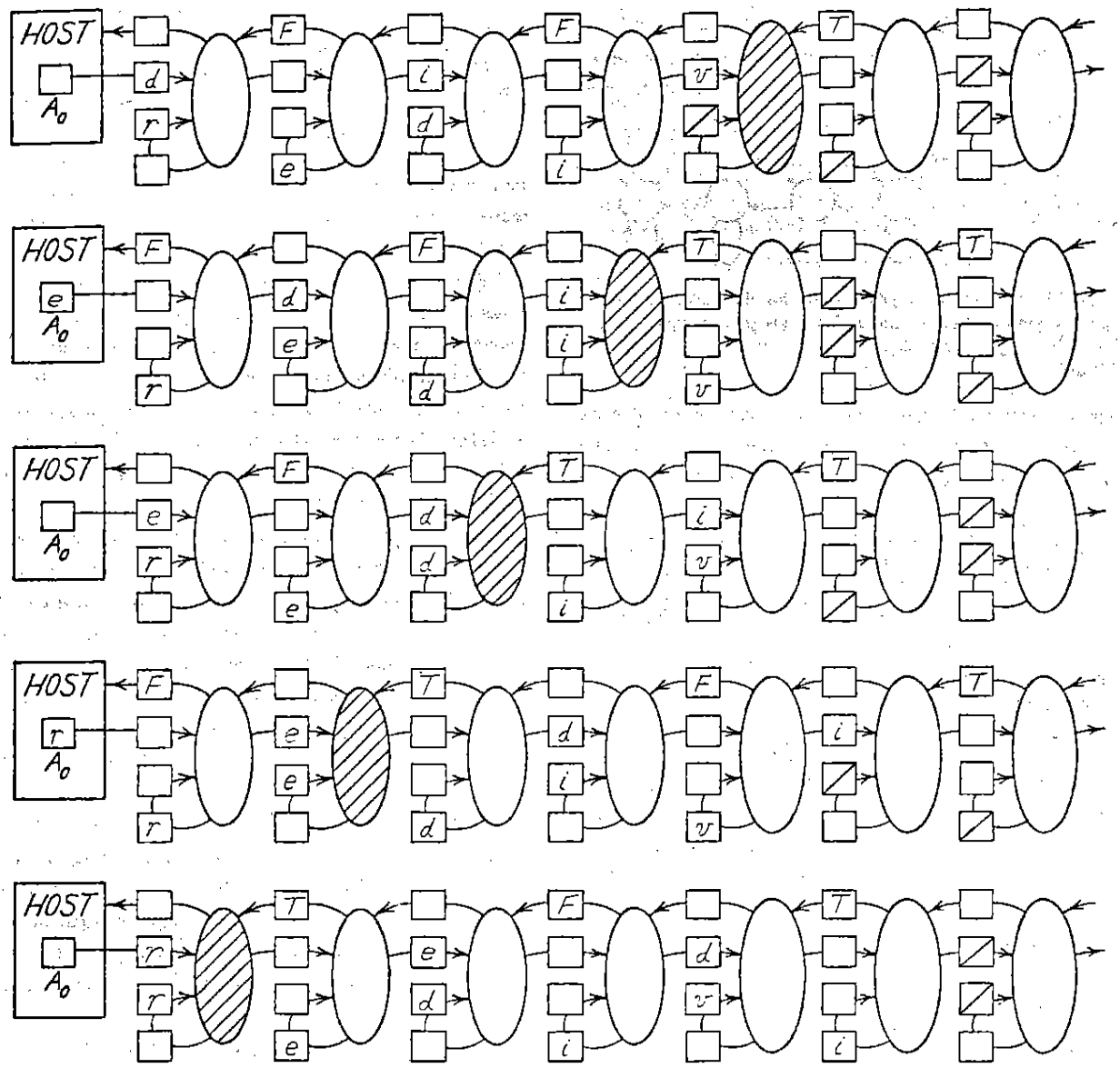


Figure 13: A sequence of configurations of the system SP showing its recognition of the palindrome "redivider".

5. A Design Methodology for Systolic Systems

The real-time palindrome recognizer from Section 4 was obtained by a three-step design process. First, an initial systolic system was designed which performed an important piece of the desired computation. Second, this systolic system was augmented with zero-weight edges to produce a synchronous system with combinational rippling. The edges were added in such a way that the Systolic Conversion Theorem could be applied to a 2-slow simulator of the intermediate system to yield the final systolic design as the third step of the design process. The subject of this section is the design methodology of augmenting a systolic system with rippling that can be eliminated by the Systolic Conversion Theorem.

In order to prejudice the designer as little as possible in his other design decisions, the method of design presented here preserves the physical organization of the original systolic system as captured by its *connection graph*, which is its communication graph viewed as an unweighted, undirected graph. There may be greater differences between two systems that share only the same connection graph than between two systems that have the same structure as defined in Section 2. Two systems with the same connection graph may have different functional elements as well as different numbers of registers on interconnections. In addition, the direction of information flow is ignored in the connection graph.

Broadcasting is a means by which information known by the host is made known to all the functional elements of a system in a single clock cycle. Broadcasting is the most common kind of global communication found in parallel systems because designers find it easy to think of controlling all processors in unison. A designer who wishes to add broadcasting to his otherwise systolic system will typically find considerable flexibility in exactly how it might be implemented. A common approach is to use a *bus*, which is a single interconnection that visits all processors and conveys the global information throughout the system. In fact, the connection graph of the system need not be disturbed if the bus is routed along any *spanning tree* of the connection graph. In our model the interconnections composing the broadcast tree can be represented as zero-weight edges in the communication graph of the synchronous system.

Even with such tricks as *precharging* the bus [11, pp. 156–157], the simple fact that information must be communicated across the system limits the performance of a broadcast because the clock period of the system must be sufficiently long to allow the global information to reach all processors. It should be apparent, however, that the first two steps of the three-step design process have been followed thus far—a systolic system has been augmented with zero-weight edges to produce an intermediate synchronous system. But can the third step succeed? If so, applying the Systolic Conversion Theorem to a 2-slow simulator of the intermediate system will produce a final systolic design whose clock period will be independent of the size of the system.

The third step need not succeed, but if broadcasting is implemented using a *breadth-first* spanning tree of the connection graph instead of an arbitrary spanning tree, it always will succeed. Let H be the connection graph of the original systolic system, and define the *depth* $d(v)$ of a vertex v in H to be the minimal number of edges in any path from v to the host. Let S be the intermediate synchronous system obtained by implementing broadcasting along a breadth-first spanning tree in H, and let G be the communication graph of S. (Thus each zero-weight *tree edge* $(u, v, 0)$ in G satisfies $d(v) = d(u)+1$, as in Figure 14.)

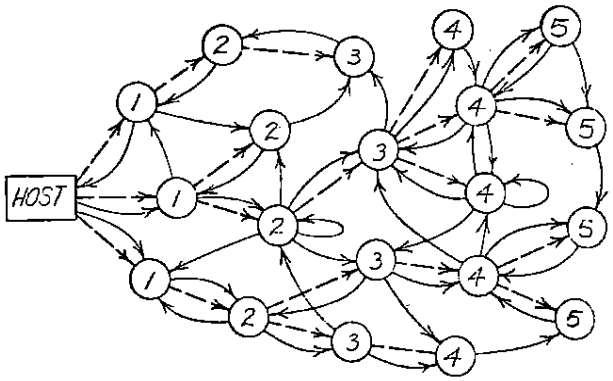


Figure 14: Broadcast from the host to all functional elements of a system can be implemented by a breadth-first spanning tree (dashed edges) of the connection graph. Each vertex is labeled with its distance from the host.

To show that the third step of the design process will work, we must demonstrate that the constraint graph $2G-1$ has no negative-weight cycles. Consider the changes in the depth d of vertices during a traversal of a directed cycle in G. By the definition of depth, traversing any edge changes d by at most one. Since traversing a zero-weight tree edge increases d by one and the net change in d around any cycle is zero, it follows that at most half the edges in any cycle of G can be tree edges. Now consider the constraint graph $2G-1$. For any tree edge in G, the corresponding edge in $2G-1$ has weight negative one. For any positive-weight edge in G, the corresponding edge in $2G-1$ has weight at least one. Because at most half the edges in any cycle of G are tree edges, no cycle in $2G-1$ has negative weight. Consequently, the Systolic Conversion Theorem can be applied to 2S to produce a systolic system which is equivalent to 2S and which has the same structure as S. Thus if broadcasting is the only form of global communication in an otherwise systolic system, it can be replaced by local communication.

Another common instance of global communication is *collection* where rippling logic runs from functional elements toward the host as in the palindrome example from Section 4. If the logic computes an associative and commutative function (such as addition, multiplication, maximum, or boolean conjunction) over values generated by the functional elements, then any spanning tree of the connection graph can be used to implement the collection. In order to obtain a systolic 2-slow simulator, a breadth-first spanning tree can once again be

employed. In contrast to the broadcasting situation, however, all tree edges are directed toward the host instead of away.

The three-step design procedure proposed in this section can be applied to any augmented systolic system as long as the rippling follows a breadth-first spanning tree in the connection graph, and all zero-weight edges go toward the host, or all go away from the host. The following theorem generalizes these conditions and can be proved by adapting the argument for broadcasting.

> **Theorem 4:** Let S be a synchronous system with communication graph G and connection graph H, and let R be some subset of the vertices of G. For each vertex v of G, define the distance $h(v)$ of v from R as the minimal number of edges in any path in H that joins v to an element of R. Suppose that every zero-weight edge $(u, v, 0)$ of G is directed away from R (*i.e.*, $h(v) > h(u)$), or alternatively that every zero-weight edge of G is directed towards R. Then there exists a systolic system which is equivalent to 2S and which has the same structure as S.

6. Further Topics

This section is a *pot pourri* of topics which include extensions both to the model and to the problems considered in this paper. The results of the first part of this section are straightforward and justification is given in sufficient detail to allow the reader to fill in the gaps. The latter part of this section contains results from a forthcoming paper based on research by the authors and Flavio Rose of MIT.

Several hosts. A natural extension to the model of synchronous systems is the inclusion of multiple, independent hosts. The multiple host model applies to problems where input streams are independent and may be skewed in time relative to one another, or where outputs are not fed back into the system as for example in many signal processing applications. The Retiming Lemma and the Systolic Conversion Theorem can be applied as long as two hosts cannot communicate in less time than the difference in their lags.

Clock skew. Moving from the discrete time domain to a continuous time domain permits the techniques used in this paper to be applied to the problem of clock skew. Since clock signals do not move across an integrated circuit chip in zero time, two processors on a chip may see the same edge of the clock signal at different times. The difference between the times that two processors see the change is called the *skew*. Across a large integrated circuit, the skew can be quite significant. Because the period of a clock is proportional to the maximum skew (see [12]), designers take great pains to buffer clock signals so that all destinations of the signal are equidistant from the clock generator. Unfortunately, the buffering circuitry may not match the system organization and can introduce complications during the layout of the circuit.

Using a continuous model for time, however, another approach can be adopted which is based on the broadcasting results of Section 5. Let the clock generator take the role of the host, and measure the distance of a processor from the clock generator in continuous time. By running clock signals away from the clock in a breadth-first spanning tree along existing data paths, the contribution of skew to the period of the clock can be reduced to the round-trip communication time between two adjacent processors. The maximum skew across the system is of no consequence—the local skew is all that matters.

Two-phase clocking. Clocking considerations arise even in the discrete time model. The clocks of many integrated circuit systems have two or more phases which act like the flood gates of canal locks. For example, a simple dynamic register consists of two halves—one half clocks data in on φ_1 and the other clocks it out on φ_2. Many design methodologies for two-phase clocking obey the rule that all signals must be clocked alternately by φ_1 and φ_2, that is, any signal clocked twice by one phase must be clocked by the other in between. It is straightforward to verify that the Retiming Lemma and the Systolic Conversion Theorem preserve this rule. Two-phase clocking of dynamic logic has another interesting property with regard to the results here. In order to preclude interference between adjacent dynamic registers, a system implemented with dynamic logic typically has two equivalence classes of computation of which only one can be used. Thus a systolic system designed in this way is a "2-slow" system to begin with, and the broadcasting results from Section 5 can be applied with no further slowdown needed.

This paper has investigated how to transform synchronous systems into systolic systems. We have shown that any synchronous system can be made systolic if we are willing to use a sufficiently large number of time steps to simulate one time step of the original system. In many cases the slowdown of the system in terms of time steps per operation is outweighed by the greater clock speed made possible by the elimination of long chains of rippling logic. Suppose, though, that we do not increase the number of time steps taken by a system at all, but just use the Retiming Lemma to improve its clock period as much as possible. With Flavio Rose, we have obtained the following results.

Minimizing rippling. Let S be a synchronous system with communication graph G. We know that if the constraint graph $kG-1$ has no negative-weight cycles, then there is a k-slow systolic simulator of S. Surprisingly, the absence of negative-weight cycles in $kG-1$ is also a necessary and sufficient condition for the existence of a synchronous system S′ such that S′ is equivalent to S and every path of length k in S's communication graph G' has positive weight. Thus the maximum amount of combinational rippling in S′ is through k functional elements. For example, consider the real-time palindrome recognizer

from Section 4. The original synchronous system P shown in Figure 7. had a 2-slow systolic simulator. Figure 15 shows the communication graph of another system which is equivalent to P, but whose clock period is less than that of P. No signal ripples through more than two functional elements.

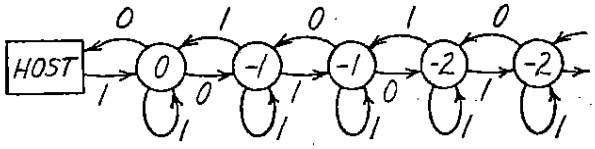


Figure 15: Communication graph of an optimized simulator for the real-time palindrome recognizer P of Figure 7. In the optimized simulator, no signal propagates through more than two functional elements in one clock period. Each vertex is labeled with its lag in this system with respect to P.

Functional elements of unequal speeds. By minimizing, as described above, the number of functional elements through which any signal can ripple during one clock tick, we are guaranteed to minimize the clock period if the combinational-logic delays through all the functional elements are equal. A more general issue addressed in the forthcoming paper is to optimize the clock period in a communication graph where vertices are each given a weight representing the delay through the functional element. The problem of determining a system with the optimal period can be reduced to a sequence of mixed integer programming problems. Although mixed integer programming is in general NP-complete, the special character of these problems permits a solution to each in $O(|V|^3+|E|)$ time. A further generalization allows a polynomial-time solution to optimization problems in which the delays between various inputs and outputs of the same functional element may be unequal.

Multiphase clocking. Clocking schemes that use more than two phases offer greater flexibility in adjusting the relative timings of the functional elements. Consider, for example, the functional element δ used in the palindrome recognizer P (see Figure 7). In any implementation of this element it is quite plausible that the delay from P_{in} to P_{out} will be considerably less that the delays from A_{in} and B_{in} to any of the outputs. Suppose the delay from P_{in} to P_{out} were only half as great as the other delays. How could we take advantage of this? Figure 16 shows a retiming of P using a three-phase clocking discipline.

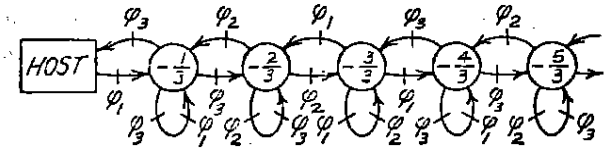


Figure 16: A fast implementation of the palindrome recognizer using three-phase clocking. Each vertex is labeled with its lag in this system with respect to P.

Signals can propagate from any P_{in} to the P_{out} of the same functional element within one phase transition time, but take two phases to travel through a functional element in any other way, and no two consecutive registers along any path are clocked on the same phase. The system shown runs 4/3 as fast as would be possible with two-phase clocking.

It is natural to ask whether the results described in the preceding paragraphs can be extended to allow the computation of optimal clock speeds for general circuits under a multiphase clocking system. The answer turns out to depend critically on the details of the clocking model in question. In one model, we can show that it is possible to test in polynomial time whether a circuit can be retimed to allow clocking at a given speed, while in another model this test is *NP*-complete. Unfortunately, there is not space here to describe the details of the two clocking disciplines in question.

7. Conclusion

System transformations which optimize clock period have been examined by others, but the transformations considered are of the basic *pipelining* kind which improve only the throughput of a system. The system is usually a one-dimensional array with an input port at one end and an output port at the other. All rippling goes from the input toward the output. By applying a restrictive version of the Retiming Lemma, registers are introduced along the length of the "pipeline" so that the clock period can be reduced (improving throughput) at the expense of skewing the timing of the two ports (worsening response time). Cohen [1] presents an imaginative methodology based on this approach.

Another system transformation which does not involve retiming is found in the Reset Theorem of [9]. This theorem states that a host can effectively reset all registers in a synchronous system to predefined values in one clock tick. Although the combinational logic in functional elements is augmented slightly, the connection graph of the system is left intact, no rippling is introduced where it didn't exist before, and the applicability of the Systolic Conversion Theorem is not affected.

The efficacy of the Systolic Conversion Theorem is due to the host computer's limited view of the synchronous system. (The idea of a host for cellular automata is apparently due to Cole [2], although Hennie [4] allows external I/O connections to all processors in his iterative arrays.) The smaller the host's view, the more flexibility there is in changing the underlying system while maintaining the view. For instance, if the host can "see" the entire system, there is no flexibility in choosing an implementation. In the context of VLSI systems, however, there seems to be ample room for optimization because the number of pins on a chip (much less than 10^3) is substantially smaller than the number of components (potentially more than 10^7).

Acknowledgment

Thanks to Jon Bentley, Dan Hoey, and Flavio Rose.

A. Proof of the Retiming Lemma

This appendix contains a detailed proof of the Retiming Lemma. The reader is cautioned that if he has not understood the intuitive explanation given in Section 2, the proof here will not enlighten him. The proof of the lemma is tortuous, and the simple ideas underlying it are obscured by many technical details and an elaborate notation. The authors apologize for being unable to provide a cleaner, shorter proof.

> Lemma 1: (*Retiming Lemma*) Let S be a synchronous system with communication graph G, and let *lag* be a function that maps each vertex v to an integer and the host to zero. Suppose that for every edge (u, v, w) in G the value $w+lag(v)-lag(u)$ is nonnegative. Let S′ be the system obtained by replacing every edge $e = (u, v, w)$ in S with $e' = (u, v, w+lag(v)-lag(u))$. Then the systems S and S′ are equivalent.

Proof: We need only show that S′ simulates S, since if the roles of S and S′ are interchanged in the statement of the lemma and $-lag$ replaces *lag*, the same proof will show that S simulates S′. First, observe that the weight of any cycle in the communication graph G' of S′ is the same as the weight of the corresponding cycle in G since the additions and subtractions of lags cancel around the cycle. Consequently, S′ is a synchronous system because S is.

Because the remainder of the proof examines the internal structure of the two systems in great detail, we introduce some terminology. A *wire* is a connection between any two components (registers or functional elements) of a system. As shown in Figure 17, any edge e of

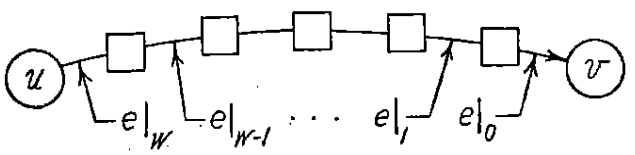


Figure 17: Division of an edge $e = (u, v, w)$ into wires.

weight w is divided by the registers along it into $w+1$ wires $e|_0, e|_1, \ldots, e|_w$, where $e|_0$ is the wire that carries inputs to the functional element at the head of e, and $e|_w$ is the wire that carries outputs from the functional element at the tail of e. For any time step t and any wire x, define $value(x, t)$ as the value asserted on x at the end of time step t, that is, after all the combinational logic has settled and after the inputs for time t have been asserted by the host, but before all the registers are clocked to begin time step $t+1$.

A wire x is a *predecessor* of another wire y if there is some functional element v (not the host) such that x carries a value into v and y carries a value out of v (so that changes might ripple through v from x to y with no registers intervening). Because S′ is synchronous, the transitive closure of the predecessor relation partially orders the wires of S′.

The wires of any system are divided into three mutually disjoint classes: register outputs, functional outputs, and host outputs. A wire of the form $e|_k$, where $k < weight(e)$, is a *register output*. A wire of the form $e|_{weight(e)}$ is either a *functional output* or a *host output* depending on whether the tail of e is an ordinary functional element or the host. A functional output may have zero or more predecessors; register outputs and host outputs never have predecessors.

The proof is based on inductive reasoning about the values asserted on the wires in S and S′. Let t_0 be the maximum lag of any vertex in G, and suppose that S is initialized in any configuration at time 0 and run with an arbitrary sequence of inputs from the host until time t_0 to arrive in some configuration c. The goal of the proof is to exhibit a configuration c' for S′ such that if S′ is started at time t_0 in this configuration, then the behaviors of the two systems will be indistinguishable from then on.

Before embarking on the inductive proof, however, we introduce the predicate $P[e'|_k, t]$ which is defined to hold for any wire $e'|_k$ of S′ and any time step $t \geq t_0$ if

$$value(e'|_k, t) = value(e|_0, t-lag(head(e))+k).$$

We must show that that P is well-defined, that is, for any wire $e'|_k$ in S′ and for any time $t \geq t_0$ the value $value(e|_0, t-lag(head(e))+k)$ is uniquely determined by the history of S from time 0 to time t. There are two cases. First, if $t-lag(head(e))+k \leq t$, then $value(e|_0, t-lag(head(e))+k)$ is well-defined provided that $t-lag(head(e))+k \geq 0$. But since $t_0 \geq lag(head(e))$ and $k \geq 0$, this follows immediately.

The second case in the demonstration of the well-definedness of P is for $t-lag(head(e))+k > t_0$. Here we must show that no data originating from the host at a time later than t can affect the value on $e|_0$ until after time $t-lag(head(e))+k$. Let the tail and head of e be called u and v, and define r as the minimum number of registers on any path from the host to u in S, and define r' as the corresponding minimum number in S′. The relationship $r' = r+lag(u)$ holds because the lag of the the host is 0 and the additions and subtractions of lags of the intermediate vertices cancel along any path from the host to u. Thus the minimal number d of registers delaying any signal from the host to $e|_0$ is given by

$$
\begin{aligned}
d &= r + weight(e) \\
&= (r' - lag(u)) + (weight(e') + lag(u) - lag(v)) \\
&= r' + weight(e') - lag(v) \\
&\geq weight(e') - lag(v) \\
&\geq k - lag(v).
\end{aligned}
$$

Consequently, no signal originating at the host at time $t+1$ or later can be propagated to $e|_0$ until at least time $t+1+d > t-lag(v)+k$, which completes the demonstration that predicate P is well-defined.

Recall that system S is in configuration c at time t_0, and the goal of the proof is to prove the existence of a configuration c' of S′ such that the behavior of S′ mimics the behavior of S from time t_0 onward. Let c' be that configuration of S′ in which, for each register output wire $e'|_k$, the register whose output is asserted on $e'|_k$ holds the value $value(e|_0, t_0 - lag(head(e)) + k)$. By the argument used to show the well-definedness of P, all the values specified are well-defined and independent of the actions of the host at any time later than t_0.

Suppose that S′ is started at time t_0 in configuration c' while S is allowed to continue from configuration c, and suppose that all host outputs in S′ are identical to the corresponding host outputs in S at all times from t_0 onward. We complete the proof by verifying in order the following assertions, where t is any time greater than or equal to t_0 in Assertions (*ii*) through (*vii*).

(*i*) $P[x, t_0]$ for every register output x in S′.

(*ii*) $P[x, t]$ for every host output x in S′.

(*iii*) If $P[x, t]$ for every predecessor x of a functional output y in S′, then $P[y, t]$.

(*iv*) If $P[x, t]$ for every register output x in S′, then $P[y, t]$ for every wire y in S′.

(*v*) If $P[x, t]$ for every wire x in S′, then $P[y, t+1]$ for every register output y in S′.

(*vi*) $P[x, t]$ for every wire x in S′.

(*vii*) All wires carrying outputs from S′ to the host at time t carry the same values as the corresponding wires in S at time t.

Assertion (*i*) follows directly from the definition of c'.

To verify Assertion (*ii*), let x be any host output in S′, and let v be the vertex at the head of the edge e' that contains x. Then, since host outputs are always the same in S and S′, it follows that

$$
\begin{aligned}
value(x, t) &= value(e'|_{weight(e')}, t) \\
&= value(e|_{weight(e)}, t) \\
&= value(e|_{weight(e')+0-lag(v)}, t) \\
&= value(e|_0, t - lag(v) + weight(e')).
\end{aligned}
$$

To verify assertion Assertion (*iii*), consider any functional output y, let e_1' be the edge on which wire y lies, that is, $y = e_1'|_{weight(e_1')}$, and let u and v be the tail and head of e_1' respectively. Suppose the predicate $P[x, t]$ holds for any predecessor $x = e_2'|_0$ of y in S′. Then for each such x we have

$$
\begin{aligned}
value(x, t) &= value(e_2|_0, t - lag(head(e_2))) \\
&= value(e_2|_0, t - lag(u)).
\end{aligned}
$$

Since the functional elements at u in S and S′ are identical,

$$
\begin{aligned}
value(y, t) &= value(e_1|_{weight(e_1)}, t - lag(u)) \\
&= value(e_1|_0, t - lag(u) + weight(e_1)) \\
&= value(e_1|_0, t - lag(v) + weight(e_1')),
\end{aligned}
$$

and hence $P[y, t]$ holds.

Assertion (*iv*) follows from Assertion (*iii*) by induction over the partial order imposed on the wires by the predecessor relation, where the base step is supplied by the hypothesis of the assertion together with Assertion (*ii*).

To demonstrate Assertion (*v*), notice for any register output $e'|_k$,

$$
\begin{aligned}
value(e'|_k, t+1) &= value(e'|_{k+1}, t) \\
&= value(e|_0, t - lag(head(e)) + k + 1) \\
&= value(e|_0, (t+1) - lag(head(e)) + k).
\end{aligned}
$$

Assertion (*vi*) follows by induction from Assertion (*iv*) and Assertion (*v*) using Assertion (*i*) as the base step.

Assertion (*vii*) is simply Assertion (*vi*) restricted to wires of the form $e'|_0$ where $head(e')$ is the host. □

References

1. Danny Cohen, "Mathematical approach to computational networks," Technical report ISI/RR-78-73, Information Sciences Institute, University of Southern California, November 1978.
2. Stephen N. Cole, "Real-time computation by n-dimensional iterative arrays of finite-state machines," *IEEE Transactions on Computers*, Vol. C-18, April 1969, pp. 349-365.
3. Michael J. Foster and H. T. Kung, "The design of special-purpose VLSI chips," *Computer Magazine*, Vol. 13, No. 13, January 1980, pp. 26-40. An early version of this paper entitled "Design of special-purpose VLSI chips: examples and opinions" appears in *Proceedings of the 7th International Symposium on Computer Architecture*, La Baule, France, May 1980.
4. Frederick C. Hennie III, *Iterative Arrays of Logical Circuits*, MIT Press and John Wiley Sons, Inc., MIT Press Research Monographs, 1961.

5. H. T. Kung and Charles E. Leiserson, "Systolic arrays (for VLSI)," *Sparse Matrix Proceedings 1978*, I. S. Duff and G. W. Stewart, ed., Society for Industrial and Applied Mathematics, 1979, pp. 256–282. An early version appears in Section 8.3 of [11].

6. H. T. Kung, "Let's design algorithms for VLSI systems," *Proceedings of the Caltech Conference on Very Large Scale Integration*, Charles L. Seitz, ed., Pasadena, California, January 1979, pp. 65–90.

7. Eugene L. Lawler, *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart and Winston, New York, 1976.

8. Charles E. Leiserson, "Area-efficient graph layouts (for VLSI)," *21st Annual Symposium on Foundations of Computer Science*, IEEE Computer Society, October 1980.

9. Charles E. Leiserson, *Area-Efficient VLSI Computation*, Ph.D. dissertation, Department of Computer Science, Carnegie-Mellon University, to appear 1981.

10. Richard L. Lipton, Donald J. Rose, and Robert Endre Tarjan, "Generalized nested dissection," *SIAM Journal of Numerical Analysis*, Vol. 16, No. 2, April 1979, pp. 346–358.

11. Carver A. Mead and Lynn A. Conway, *Introduction to VLSI Systems*, Addison-Wesley, Reading, Massachusetts, 1980.

12. Charles L. Seitz, "System timing," in *Introduction to VLSI Systems* by Carver Mead and Lynn Conway, Addison-Wesley, Reading, Massachusetts, 1980, pp. 218–262.

13. Clark D. Thompson, *A Complexity Theory for VLSI*, Ph.D. dissertation, Department of Computer Science, Carnegie-Mellon University, 1980.