LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

A Design Methodology for Stoppable Clock Systems

Computation Structures Group Memo 240
August 1984

Willie Y-P Lim

This research was supported by the Department of Energy under grant no. DE-AC02-79ER10473

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

A Design Methodology for Stoppable Clock Systems[1]

Willie Lim
MIT Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139

September 4, 1984

Abstract

Many approaches have been developed for designing large, high performance computer systems. Classical synchronous approaches are susceptible to synchronization problems at the clock pulse level. Newer asynchronous approaches, on the other hand, avoid such problems but are expensive to implement. This paper proposes a compromise approach that builds on the well developed synchronous system design techniques and at the same time avoids the clock pulse level synchronization problems. In this approach, a system has a totally synchronous core with a "stoppable" clock and uses an asynchronous interface for external communication. With the clock not running, the asynchronous interface receives and sends information in the form of packets, setting up the proper input values and initial state for the synchronous core. The clock is then started and the synchronous core behaves as a sequential state machine initialized to the proper state and subjected to the proper input values. When the core is done with its computation, the clock is stopped and the process is repeated. A methodology for building such systems is presented in this paper.

Index terms — Asynchronous communication, clocks, design methodology, multi-processor systems, packet communication, state machines, synchronization.

---

[1]This research was supported by the Department of Energy under grant number DE-AC02-79ER10473 and the National Science Foundation under grant number MCS-7915255.

1. Introduction

Large computer systems, capable of exploiting a high degree of parallelism, are being developed in the areas of artificial intelligence [6], vision [1,2], and data flow [5]. The number of processors in such systems is typically in the range of a few hundreds to several thousands. Such a large number of processors poses several problems in the design and implementation of such systems. As the number of processors increases the distances separating the processors become significant. It is not the magnitude of actual physical distances that is important but rather the relative difference in distances between the longest and shortest communication links between processors. Hence it does not matter if the processors are all in the same chip, printed circuit board, rack or room. Whatever communication discipline is used and however large the number of processors involved, there must be no large fluctuations in the transmission delays. It is impractical to do this by the fine tuning, through careful design, of each individual inter-processor link in the system. A more general design have to be used for all links. It is important to note that the communication discipline used does influence the internal structure of the processors.

Conventional approaches to processor design have been centered on synchronous state machines, i.e. a processor driven by a single clock. Such approaches are not feasible for implementing large, high performance systems. This is due in part to the fact that an extremely slow clock has to be used to insure that all parts of the system, however far apart, will sense the clock pulses for synchronous operation. The next obvious choice would be to use multiple clocks in the system, letting say each processor have its own clock. This introduces another problem. Since the data sent by a processor is only dependent on its own clock and hence is totally independent of the clock at the destination processor, the arriving data have to be synchronized with respect to the destination's clock. It has been found [3,4] that the receiving party can at times fail to detect all the arrivals of such externally generated data leading to what is termed a *synchronizer failure*. Techniques have been developed to avoid synchronizer failures [10,11,12]. Such techniques include the introduction of special clocks where the generation of clock pulses can be stopped by control signals from the system being clocked. Such clocks are frequently referred to as *pausable* or *stoppable* clocks. Another approach to avoid the need for synchronization is to use a totally asynchronous or "clockless" system. An example of such systems is the self-timed system of Seitz [11]. If such a system is composed entirely of primitive self-timed elements which are similar in functional complexity to logic gates, the cost would be prohibitive.

This paper describes a methodology for designing large digital computer systems that are based on stoppable clocks and on the communication principles of asynchronous systems. Such systems termed *stoppable clock systems*, are not new. They have been used by Seitz [11] and Pčchoucčk [10].

The methodology is particularly well suited for building systems where the component processors are activated only when necessary. In such systems, the processors' clocks need be running only when the actual processing of the data is taking place, i.e. after the data have arrived at the destination processor. Once the data are processed, the clock can be turned off. No synchronization is necessary since the clock is not running when the data arrive. As there are subtleties in how the clock can be started and stopped as well as in how such systems can be used, the methodology enforces a discipline in the design of the communication links and the processors.

A stoppable clock system is a more general form of a sequential state machine. This is illustrated in Section 2 which defines the model for stoppable clock systems. The methodology, discussed in Section 3, is based on the model. Three simple examples are given in Section 4 to illustrate the use of the methodology. This is followed by the conclusion of the paper presented in Section 5.

2. A Model for Stoppable Clock Systems

The model presented here is an extension of the classical finite state machine model of synchronous machines [7]. The set of values that an input can have is extended from the set $B = \{0, 1\}$ of Boolean values to the set $C = \{0, \uparrow, 1, \downarrow\}$ where $\uparrow$ and $\downarrow$ represent the low-to-high and high-to-low signal transitions, respectively. If a value is in the set C, then it is termed a ***signal value***. All signal inputs cycle through the four values in C. Since signal values are used instead of Boolean values and a clock is explicitly included in the model, the term *finite state system* is used instead of finite state machine. When the clock is running, all signals that are used in computing the output values and the next state of the system must be in the set B. It is assumed that the system will only process those inputs that have stable values when the clock is running. That is, all the other inputs are treated as "don't cares".

Before the clock of a stoppable clock system can be started, the system must be set in the proper initial state. Since the initial state can be the final state of the last state transition (i.e. the last time the clock is run), the term *rest state* is used instead. Two special functions — the *clock control function* f for starting the clock, and the *state initialization function* $\mathbf{I}$ for setting the initial state of the system — are used in the model. These are the only functions that can be computed when the clock is not running. They use inputs with signal values and produce Boolean outputs. Let I be the set of vectors of signal values for the inputs and R the set consisting of the initial and the final states, then the functions have the mappings:

$$f : I \times R \rightarrow B$$

$$\mathbf{I} : I \times R \rightarrow S$$

The following is the definition of a stoppable clock system obtained by adding f and I to the tuple defining the state machine that forms the synchronous core of the system.

Definition 1: A *stoppable clock system* is a finite state system (S, I, O, **T**, **O**, I, f) where

S is the set of states,

I is the set of input signal vectors,

O is the set of output vectors,

$\mathbf{T} : I \times S \rightarrow S$, is the *state transition function*,

$\mathbf{O} : I \times S \rightarrow O$, is the *output function*,

f: $I \times \mathcal{R} \rightarrow \mathcal{B}$, is the clock control function, and,

$\mathbf{I} : I \times \mathcal{R} \rightarrow S$, is the state initialization function, with

$\mathcal{R} \subseteq S$ being the set of rest states.

3. The Methodology

Before going into the details of the methodology, a couple of terms are defined. Each part of the system that runs on a single clock is termed a *module*. Since the number of modules in a system and the distances between modules can be large, asynchronous communication is used. The type of communication adopted is packet oriented in the sense that information is transmitted in the form of streams of entities termed *packets*.

The methodology covers the following issues — the protocol for inter-module communication, functional elements used for implementing stoppable clock systems, and implementations of the clock and the state initialization functions.

3.1. The Communication Protocol

Each transmission of a packet from a source module to a destination module proceeds as follows. The source sends a packet of information to the destination. Arrival of the packet at the destination is indicated via an event, termed a *ready event*, that is initiated explicitly or implicitly by the source. Explicit initiation of that event is done by sending a special signal to the destination. In implicit initiation, the occurrence of the event is derived from the arrivals of the data signals. On detecting the occurrence of the ready event, the destination accepts the packet and initiates another event termed an *acknowledge event* that indicates to the source that the packet has been accepted. The source must not send another packet until it has detected the acknowledge event.

The data carried by a packet are sent in parallel through the data lines. If the ready event is to be explicitly initiated, a special signal termed the

Input Pairs	Output
(1, 1), (1, 0), (1, ↑), (1, ↓)	1
(↑, ↑), (↑, 0)	↑
(0, 0)	0
(↓, ↓), (↓, 0)	↓
(↑, ↓)	undefined

Table 1: Functionality of an OR Gate

ready signal is used. This signal must not arrive at the destination sooner than any of the data signals. If this cannot be ensured, then the implicit ready signal scheme must be used. In this scheme, the ready signal is encoded in the data. The "M out of N" encoding method, where the data and ready signals are sent as M concurrent transitions over N signal lines with $N > M$, is commonly used for this purpose. If $M = 1$ and $N = 2$, the signaling convention is referred to as dual-rail signaling [11]. The "M out of N" form of signaling is more reliable than the explicit ready signaling scheme. However the circuitry for detecting (or deriving) the ready signal is more complex. In both schemes, the acknowledge event is conveyed via an *acknowledge signal.* To avoid problems with initialization of the detection circuits, it is required that all signals return to zero, i.e. the transmission of a packet is always preceded by all the data lines being reset (viz. the "spacer" state in dual rail signaling) as well as the acknowledge signal be deasserted. After the source has received the acknowledge signal, it resets all the data and ready (if used) lines and wait for the acknowledge line to reset. The destination on seeing the data and ready lines being reset will reset its acknowledge line and will set up its input circuits to receive the next packet. The source on seeing the deassertion of the acknowledge signal will then send the next packet and the cycle is repeated for subsequent packets.

3.2. Functional Elements of a Stoppable Clock System

The functional elements used for implementing a stoppable clock system are:

1. logic gates — AND, OR and NOT gates,
2. stoppable clocks,
3. arbiters,
4. I/O ports.

The functionality of the OR and AND gates are defined in Tables 1 and 2, respectively. Note that the outputs of the gates are undefined (i.e. not in the set C) if the input signal values are transitions going in opposite directions. It is the occurrence of such illegal signal values that causes synchronizer failures.

Input Pairs	Output
(0, 0), (0, 1), (0, ↑), (0, ↓)	0
(↑, ↑), (↑, 1)	↑
(1, 1)	1
(↓, ↓), (↓, 1)	↓
(↑, ↓)	undefined

Table 2: Functionality of an AND Gate

Even though stoppable clocks, arbiters, and the I/O ports can be built using the logic gates, they are treated separately to emphasize their uses in stoppable clock systems. Note that the arbiters and I/O ports can only be used for implementing f and **I**.

Functionally, a stoppable clock has a *stop* and a *run* state which can be set by an input to the clock. When the clock is in the run state, clock pulses are generated. Clock pulse generation stops when the input is set to low. Correct implementation of such clocks require that the input be set low only when the clock is not about to generate a clock pulse. If the input signal is set low well before the generation of the next clock pulse, then clock pulse generation can be successfully inhibited. Furthermore if the input signal is set low well after the next clock pulse is generated, then the generation of that clock pulse cannot be inhibited. Unpredictable clock behavior results when the input signal is set low just as the clock pulse is being generated. Knowing the clock period and the time of generation of the last clock pulse, the clock input can be controlled such that it will be set low well before the next clock pulse is generated. It is assumed that such a constraint on the clock input is always met.

An arbiter is a two-input and two-output mutual exclusion device which produces exactly one high output when one or both of its inputs is high. The output that is high indicates which input is the 'first' to become high. The two inputs to the arbiter, R0 and R1, carry the request signals while the outputs, G0 and G1, are for the grant signals generated in response to the request signals. It is assumed here that the outputs of the arbiters are always logically defined.

An input or an output port is a buffer capable of holding one packet. Figure 1 shows both kinds of ports. For the input port, the signals at the input side are asynchronous. There is a set of data lines and two control lines at the output side of the port. One of the control lines (the FULL line) is for indicating if the buffer is full while the other (the READ line) is for clearing the buffer, i.e. setting it to the empty state. A packet can only be received by an input port if it is empty. It will not send an acknowledge signal back to the source if it is still full. For the output port, the data lines and control signals are on the input side while the asynchronous signal lines are on the output side. One of the control lines (the WRITE line) is used for

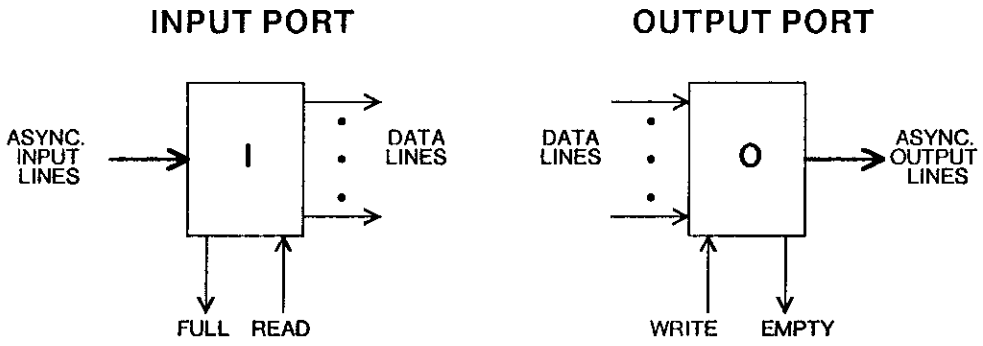

Figure 1: Input and Output Ports

writing to the buffer while the other (the EMPTY line) is for indicating if the buffer is empty or full i.e. has been read or not. The necessary asynchronous signaling for loading a packet to an input port or unloading a packet from an output port is performed by the port itself. Note that the READ and WRITE signals are synchronous since they are generated by the system when the clock is running.

3.3. Implementing f and I

The FULL signals of the input and output ports are used for implementing f and I. Note that since the ports are loaded or unloaded asynchronously, the FULL/EMPTY signals are asynchronous signals. Thus it is important that the sensing of these signals be done only when the clock is not running. Even though a system can selectively activate and deactivate certain ports, for simplicity it is assumed here that all the ports are active i.e. all the FULL/EMPTY signals are used for implementing f and I. Hence f is a function mapping the FULL/EMPTY signals to a Boolean value. It essentially detects the conditions (expressed in terms of the FULL/EMPTY signals) under which the clock is started. Note that the condition for stopping the clock is strictly determined by the system, i.e. the clock is stopped synchronously.

The initial state of a system when the clock is started has two components — the state of the system when the clock is started and the state of the ports. The former is the same as the state of the system when the clock was stopped. However that component alone may not be sufficient to take into account the state changes of the ports when the clock is not running. Thus the second component is needed to completely specify the initial state. In the methodology presented here, the function I is restricted to map from the set of FULL/EMPTY signals to a Boolean vector. This function essentially identifies the condition that starts the clock while f merely detects that condition.

Note that the signals representing the values of f and I are asynchronous

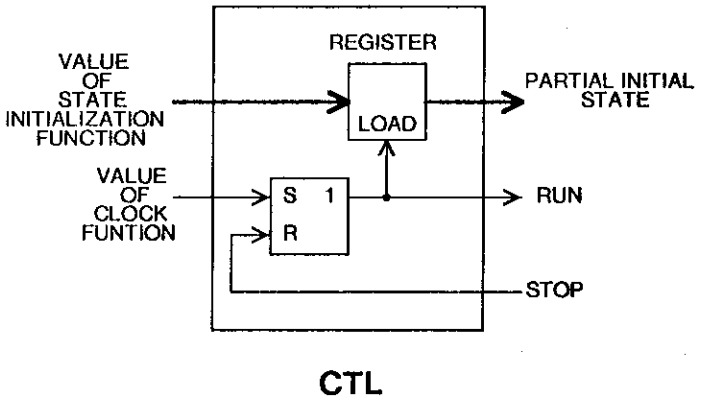

Figure 2: Control Circuit for f and $\mathbf{I}$

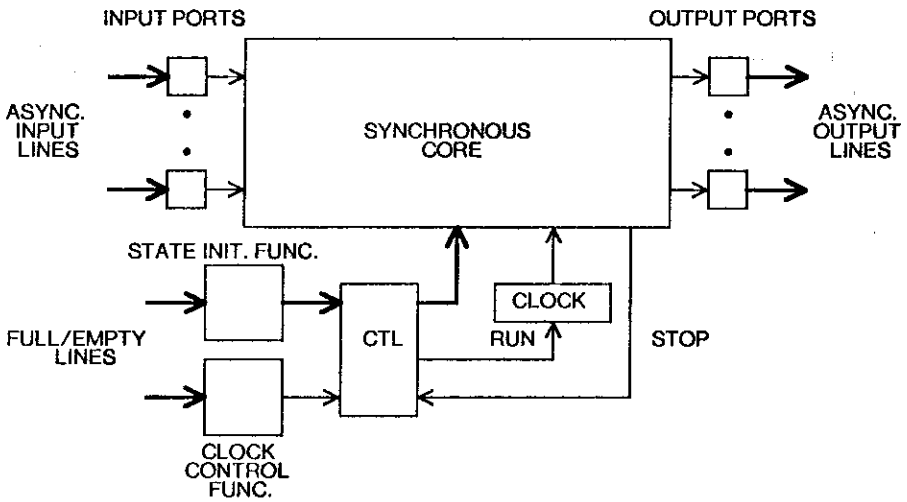

Figure 3: A Stoppable Clock System

since they are produced by asynchronous inputs. It is important that their values do not change while the clock is running. Thus the generation of new values for these functions can only occur when the clock is not running. This basically means that the hardware that implements these functions must have enabled/disabled modes such that when the clock is running, the hardware is set in the disabled mode which prevents further changes in the FULL/EMPTY signals from affecting the current values of the functions. When the clock is stopped, the hardware is enabled allowing new values for the functions to be computed. A circuit employing edge-triggered storage elements for doing this is shown in Figure 2. The values for f and **I** are applied to the inputs of the circuit and the outputs are Boolean signals that remain unchanged while the clock is running. The RUN output is used to start the clock. The system stops the clock by asserting the STOP line. Figure 3 shows how the control circuit is used in a stoppable clock system.

4. Some Examples

Three examples are used to illustrate how f and **I** can be implemented. The first two are simple modules that basically operate on a "demand" basis, i.e. they only get activated when they receive input packets. A more interesting case is the third example where the module is always active. That is, it always have something to do even when there are no input packets. The arrival of input packets causes the internal activities to be interrupted.

4.1. A Simple Module

Consider a simple one-input, one-output module. The module absorbs a packet, performs some computation with it and sends out a packet at the end of the computation. For this module, the clock is started when the input port is FULL and the output port is EMPTY. Once the input packet is read, the READ signal is sent clearing the input port. The clock is stopped after the output packet is written to the output port. For this case, f is merely the AND of the FULL and EMPTY signals. When the clock is started, it means that the input port is full and the output port is empty. Hence there is no need for computing additional state information and thus **I** need not be implemented.

4.2. A Two-Input Server

Figure 4 shows the circuits for implement f and **I** for a module acting as a server to two other modules. Each user module sends a packet to the server requesting for service. The server tends to the requests on a first-come first-serve basis. When a request is served, packets are sent out via the output ports. An arbiter is used for enforcing the first-come first-serve policy. The FULL signals of the input ports are connected to the inputs of

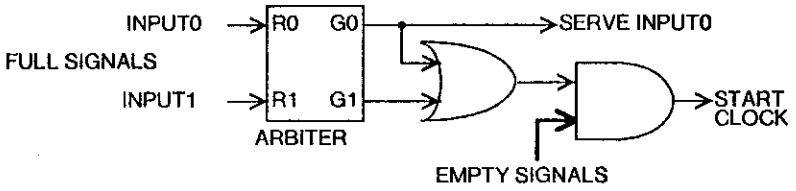

Figure 4: A Two-Input Server

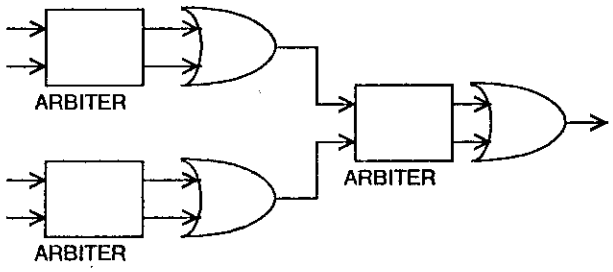

Figure 5: A Tree of Arbiters

the arbiter. The clock function f is implemented by taking the OR of the arbiter's outputs and ANDing that with the EMPTY signals of the output ports. For I, the output values produced are just those of the arbiter's outputs. The system uses these two values to determine which input port to serve once the clock is started. Just before the clock is started, the G0 output value of the arbiter is stored in CTL. Since the START CLOCK input is high only when one of the arbiter outputs is high, a low (high) value of the G0 output will indicate that INPUT0 (INPUT1) is the input to serve. The hardware in CTL will then disable the sensing of the arbiter outputs so that processing of the request can proceed. Once the packet at that input port is processed the port is reset by setting the READ input and hence that port can accept another packet while the current request is being processed. After processing the current request, the clock is stopped and sensing of the arbiter outputs is enabled by asserting the STOP line.

The server can be generalized to serve more than two inputs by using a tree of arbiters. Figure 5 illustrates how f can be implemented for this case. In general, an input to the arbiter at an interior node of the tree is the OR of the outputs of one of the two "descendant" arbiters. Multiplexors are used for implementing I which essentially identifies the port to be served. This is done by tracing a path from the root of the tree to the input port at the leaf. All arbiters on that path have a matching pair of input and output that are both asserted.

4.3. An Operation Module

Figure 6 shows how f and I can be implemented for a one-input and

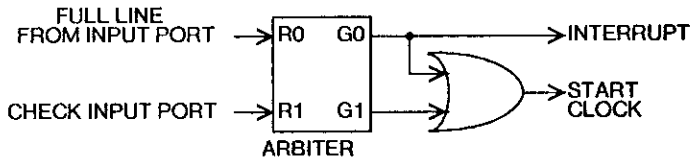

Figure 6: An Operation Module

one-output operation module. Unlike the case discussed in Section 4.1, this module always has something to do and the arrival of a packet merely causes those activities to be interrupted. For simplicity assume that the output port is always empty when the module writes to it. The module "checks" the status of its input port every now and then. An arbiter is used to implement f. As the clock is stopped, a signal is sent by the module to an input of the arbiter. The other input of the arbiter is driven by the FULL signal of the input port. Hence the arbiter outputs will indicate if there is a full input port. In either case the clock is always restarted. Thus f is just the OR of the arbiter's outputs. The function **I** is similar to that discussed in Section 4.2. When the arbiter outputs indicate that the input port is full, the module will start in a state indicating that there is an interrupt and will then service that interrupt accordingly. That interrupt mode will not be activated if the input port is not full. The time interval between "checking" of the port is variable and can be as short as one clock period. Hence the module can vary the frequency of "checks" accordingly to the situation. Furthermore this module can be generalized to handle more input ports.

The arbiter in this module performs a function very much like that of a synchronizer. However it must be noted that in this case, all signals that are sensed when the clock is started is always logically defined. This is unlike the delay-based synchronizer where there is no guarantee that its output is logically defined when sensed. The scheme has some similarities with the pausable clock scheme [10,11,12] where the synchronizer has a special output to indicate if it is in the metastable state and the clock is paused until the synchronizer is out of the metastable state. For the scheme presented here, the synchronizer being in the metastable state is equivalent to the arbiter not being able to assert one of its output in response to the simultaneous appearance of signals at it inputs. The clock is stopped in both cases. It is extremely rare that the clock will be stopped for such a long time as to have adverse effects on the performance the module [8,9].

5. Conclusions

The design methodology proposed here exploits the advantages of both synchronous and asynchronous systems. It does not require a radically new approach but rather builds on the familiar and extensive experience that

has been accumulated in the design of synchronous systems. Hence as far as the core sequential machine that drives the stoppable system is concerned, classical synchronous system approaches can still be used. The methodology avoids the problem of clock pulse level synchronization that plagues classical synchronous systems by adopting some of the principles developed for asynchronous systems. This is done at a small cost compared to that incurred when a totally asynchronous approach is used. The approach presented in this paper offers a good compromise between the two classical approaches.

6. Acknowledgments

The author would like to thank Bill Ackerman, Jack Dennis, Tam-Anh Chu, Jim Vellenga, and Tom Wanuga for their comments and suggestions.

REFERENCES

[1] Ballard, D.H., Hinton, G.E., and Sejnowski, T.J., "Parallel Visual Computation," *Nature*, vol. 306, 3 November, 1983, pp. 21-26.

[2] Brady, M., "Parallelism in Vision," *Artificial Intelligence*, vol. 21, no. 3, September 1983, pp. 271-283.

[3] Chaney, T.J., Ornstein, S.M., and Littlefield, W.M., "Beware the Synchronizer," *Digest of papers — COMPCON '72*, IEEE Computer Society, September 1972, pp. 317-319.

[4] Chaney, T.J., and Molnar, C.E., "Anomalous Behavior of Synchronizer and Arbiter Circuits," *IEEE Transactions on Computers*, vol. C-22, no. 4, April 1973, pp. 421-422.

[5] Dennis, J.B., "Data Flow Supercomputers," *Computer*, IEEE, vol 13, no. 11, November 1980, 46-54.

[6] Hillis, W.D., "The Connection Machine," *A.I. Memo 646, Artificial Laboratory*, Massachusetts Institute of Technology, September 1981.

[7] Kohavi, Z., **Switching and Finite Automata Theory**, McGraw-Hill Book Company, New York, 1970.

[8] Lim, W.Y-P., "Performance Criteria Related to the Synchronization of Clocked Systems," *M.S. Thesis*, Department of Computer Science, Washington University, St. Louis, Missouri, August 1979.

[9] Lim, W.Y-P., and Cox, J.R., Jr., "Clocks and the Performance of Synchronizers," *IEE Proceedings on Computers and Digital Techniques, Part E*, vol. 130, no. 2, March 1983, pp. 57-64.

[10] Pěchoucěk, M., "Anomalous Response Times of Input Synchronizers," *IEEE Transactions on Computers*, vol. C-25, no. 2, February 1976, pp. 133-139.

[11] Seitz, C., "System Timing," Chapter 7 of **Introduction to VLSI Systems**, by C. Mead and L. Conway, Addison-Wesley, Reading, MA, 1980.

[12] Stucki, M.J, and Cox, J.R., Jr., "Synchronization Strategies," *Proceedings of the Caltech Conference on VLSI*, California Institute of Technology, January 1979, pp. 375-393.