Implementing Sequential Machines as Self-Timed Circuits

Ilana David, Ran Ginosar, *Member, IEEE*, and Michael Yoeli

Abstract— **A self-timed finite state machine is described. It is based on a formally-proven, efficient implementation of self-timed combinational logic and a self-timed master–slave register. Temporal behavioral constraints are formalized, and the system is shown to abide by them. The synthesis method is algorithmic, and serves as an automatic compiler of self-timed FSM's. The method is compared with other approaches.**

Index Terms— **Asynchronous systems, combinational logic, delay-insensitive, finite state machines, master–slave register, self-timed.**

I. Introduction

SELF-TIMED logic provides a method for designing asynchronous hardware circuits such that the correct behavior of the circuit depends neither on the speed of its components nor on the delay along its communication wires. In addition, the circuits can generate a completion signal. The advantages of self-timed logic, as compared with globally clocked logic, are discussed, e.g., in [12].

Digital circuits are conveniently classified according to levels of increasing complexity. At the lowest level, combinational logic implements Boolean functions. Next come sequential finite state machines, which can be described by regular expressions and other equivalent representations. More complex designs usually consist of networks interconnecting combinational logic and finite state machines in various forms. Traditionally, we have been accustomed to synthesizing computer architectures mostly out of these two types of building blocks.

While synchronous implementations of such digital circuits prevail widely, self-timed systems present an attractive alternative. However, self-timed design is not as straightforward as synchronous circuits, as such designs avoid the relaxing assumptions and external enforcement of timing available with synchronizing clocks. In an attempt to approach those problems formally and efficiently, we have designed self-timed combinational logic and finite state machines in a formal manner, while stressing efficient implementation. As a result, those two levels of building blocks are now available for safe construction of digital systems of higher complexity.

In a companion paper [3] we have described a method for implementing any family of Boolean functions as an efficient self-timed circuit *(CL)*. This paper proposes a general method for efficiently implementing finite-state machines as self-timed systems. We start by defining and implementing a self-timed master–slave register. Subsequently, the self-timed finite state machine is constructed by connecting a CL module with a self-timed master–slave register.

The FSM is specified by a state table, similar to the specification of a Mealy-type synchronous FSM [6]. However, the synchronizing clock is replaced by sequences of events. We also impose a set of temporal behavioral constraints on the FSM, extending the approach taken by Seitz [12] for CL and as modified by us [3]. The inputs and outputs of the circuit are ternary. The FSM, similar to our CL [3], produces a completion signal. The correctness of the FSM can be proven formally; thus, it can serve as a "correct by construction" building block for automatic system synthesis.

The next section specifies sequence constraints and the construction of self-timed master–slave registers. Section III describes the sequence constraints of the FSM, and Section IV discusses its construction. In Section V we compare our approach with others.

II. Self-Timed Master–Slave Register

Our self-timed master–slave (MS) register operates similarly to a conventional master–slave register: it first stores the data at its input and only later outputs the stored data. However, it is not controlled by a clock, rather by the sequence of events.

The ternary n-stage self-timed master–slave register (MS) is defined as follows:

1) The MS has n three-valued inputs $\hat{Y}_1, \cdots, \hat{Y}_n$ and n three-valued outputs $\hat{y}_1, \cdots, \hat{y}_n$.
2) Each input and output may assume any value from the set $\{0, 1, U\}$. We refer to U as the "undefined" value and to $0, 1$ as the "defined" values.
3) The sequential behavior of the MS and its environment is constrained by the following cycle of activities: The E_i's are environment (domain) constraints; the S_i's are network (functional) constraints. $E0$ is the initialization of the MS; thereafter, each cycle starts with $E1$ and terminates with $S4$.

Manuscript received July 13, 1989; revised April 23, 1991. The work was supported in part by Technion VPR fund—Elron-Elbit Electronics Research Fund.

I. David is with the Department of Electrical Engineering, Technion–Israel Institute of Technology, Haifa 32000, Israel.

R. Ginosar is with the Department of Electrical Engineering and the Department of Computer Science, Technion–Israel Institute of Technology Haifa 32000, Israel.

M. Yoeli is with the Department of Computer Science, Technion–Israel Institute of Technology Haifa 32000, Israel.

IEEE Log 9103033.

0018-9340/92$03.00 © 1992 IEEE

$E0$.	All inputs are set to "undefined" and all outputs are set to the initial-state defined value.
$E1$.	Some (but not all) inputs become defined.
$S1$.	All outputs remain defined.
$E2$.	All inputs become defined.
$S2$.	All outputs become undefined. The value of the input vector is saved in the register; an acknowledgment output is produced.
$E3$.	Some (but not all) inputs become undefined.
$S3$.	All outputs remain undefined.
$E4$.	All inputs become undefined.
$S4$.	All outputs become defined and their value equals to the stored value of the inputs; an acknowledgment is produced.

The circuit that implements the self-timed master–slave register (MS) is shown in Fig. 1. All the three-valued (input, output, and internal) lines are implemented by double-rail code as in the companion paper [3]. For example, $\hat{Y}_i$ is represented by Y_i^0, Y_i^1. We now show that the MS circuit (Fig. 1) obeys the above sequential constraints. The following discussion is given in lieu of an elaborated formal proof, of the kind developed in [3].

In the circuit, line "A" becomes 1 once all inputs $\hat{Y}_1, \hat{Y}_2, \cdots \hat{Y}_n$ become undefined, it becomes 0 once all inputs are defined, and it retains its previous value otherwise. Line "B" behaves similarly with respect to the outputs $\hat{y}_i$. The $\hat{w}_1, \cdots, \hat{w}_n$ lines save the input values, to be stored in the register.

Initially, all Y_i^0, Y_i^1 equal 0, and all $\hat{y}_i$'s are defined. It follows that "A" equals 1, "B" equals 0, and all w_i^0, w_i^1 equal 0 ($E0$). Line "W" equals 1, signaling to the environment that the inputs may now become defined.

When *some* (but not all) inputs become defined ($E1$), "A" remains 1, y_i remain defined, "B" remains 0, all w_i's remain 0, and the output values do not change ($S1$).

Once *all* inputs have become defined ($E2$), line "A" becomes 0. Consequently, all the C-elements at the outputs have their two inputs equal 0, causing their outputs to become all 0 ($S2$). This makes line "B" turn 1, which "opens" the input C-elements. As a result, the w_i's get the value of the Y_i's ($S2$). Consequently, line "W" becomes 0, signaling to the environment that the inputs may become undefined.

When *some* (but not all) inputs become undefined ($E3$), "A" remains 0, and the output values remain 0 ($S3$). "B" remains 1, so the w_i's retain their values.

When *all* inputs become undefined ($E4$), line "A" becomes 1. The C-elements at the outputs pass the w_i-values to their outputs; thus, the outputs become defined and their values equal the stored value of the inputs ($S4$). As a result, line "B" becomes 0, and then all the w_i's become 0. Line "W" becomes 1, signaling that the inputs can become defined again.

To summarize, the self-timed MS register informs the environment when the inputs may be applied and when they may be removed by means of the "W" line. Essentially, "W" indicates when the slave part of the MS register has completed its transition. Given that the inputs are undefined, only once "W" becomes 1 may the inputs become defined. Then they have to remain defined as long as "W" has not changed to 0.

In order to set the circuit to its initial state, we must be able to preset/clear the CE's. For this purpose, we add "preset" and "clear" inputs to the standard CE circuit. The logic diagram of CE with and without "preset" and "clear" inputs is shown in Fig. 2.

III. Definition and Sequential Constraints of Self-Timed Finite-State Machine

Similarly to the CL, the FSM waits for all its inputs to become defined before operating. Once all inputs have arrived, the FSM switches states and produces the corresponding outputs. Subsequently, the inputs may be removed. Note that this is different from the general *fundamental-mode* asynchronous sequential machines [6], where a single change in one of the input lines triggers a state transition. In the version of ST-FSM described here, all inputs must first become undefined and then become defined between two successive state transitions. In this respect, our ST-FSM resembles synchronous FSM's (where the clock is replaced by a certain sequence of events) more than asynchronous ones.

The self-timed FSM is shown in Fig. 3 and is defined as follows:

1) The ST-FSM consists of an interconnected CL (as defined in the companion paper [3]) and a MS (as defined in Section II).
2) The ST-FSM has n three-valued inputs $\hat{I}_1, \cdots, \hat{I}_n$, m three-valued outputs $\hat{O}_1, \cdots, \hat{O}_m$, k three-valued "next-state" lines $\hat{Y}_1, \cdots, \hat{Y}_k$ and k three-valued "present-state" lines $\hat{y}_1, \cdots, \hat{y}_k$.
3) Each input, output, and state line may assume any value from the set $\{0, 1, U\}$.
4) The sequential behavior of the ST-FSM and its environment is constrained by the following cycle of activities: The E_i's are environment (domain) constraints; the S_i's are network (functional) constraints. $E0$ is the initialization of the ST-FSM; Each cycle starts with $E1$ and terminates with $S4$.

E0	All inputs and all outputs are set to "undefined." The "next-state" is set to undefined, the "present-state" is defined and its value equals the initial state.
$E1$.	Some (but not all) inputs become defined.
$S1$.	All outputs remain undefined, the "next-state" remains undefined and the "present-state" remains defined.
$E2$.	All inputs become defined.
$S2$.	All outputs and the "next-state" become defined. Their value is determined by the CL functions. The "present-state" becomes undefined. The value of the

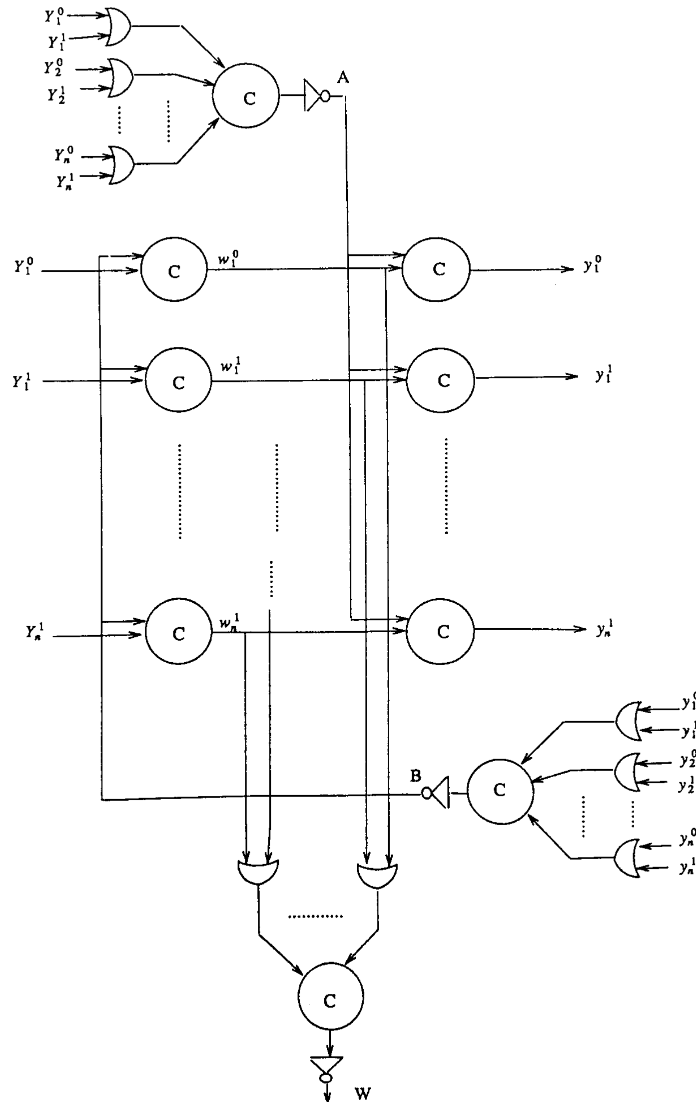

Fig. 1. Self-timed master-slave register. When all the inputs Y become defined, the value is saved inside the register (w), and the outputs y become undefined; when all the inputs become undefined, all the outputs become defined, assuming the stored value of the inputs.

"next-state" is saved in the register. An acknowledgment signal is produced.

$E3$. Some (but not all) inputs become undefined.

$S3$. All outputs and the "next-state" remain defined. The "present-state" remains undefined.

$E4$. All inputs become undefined.

$S4$. All outputs and the "next-state" become undefined. The "present-state" becomes defined and its value equals the stored value of the "next-state." An acknowledgment signal is produced.

Similar to the MS register, the FSM also employs an acknowledgment signal to notify the environment when the inputs may be applied and when they may be removed, as discussed below.

IV. Implementation and the Correct Behavior of the ST-FSM

In order to implement a self-timed FSM, we connect a self-timed double-rail combinational logic circuit (CL) with a self-timed double-rail master–slave register (MS), as shown in Fig. 3. We use the "W" line of the MS (Fig. 1) as the acknowledgment signal, to provide handshaking with the environment.

It can be formally proven that the FSM obeys the sequential constraints formulated above, by reasoning techniques similar

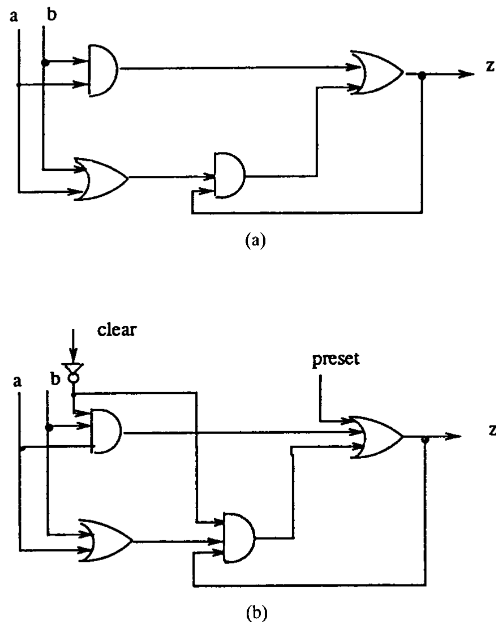

Fig. 2. A gate-level implementation of the C-element: (a) Without preset/clear inputs, (b) With preset/clear inputs.

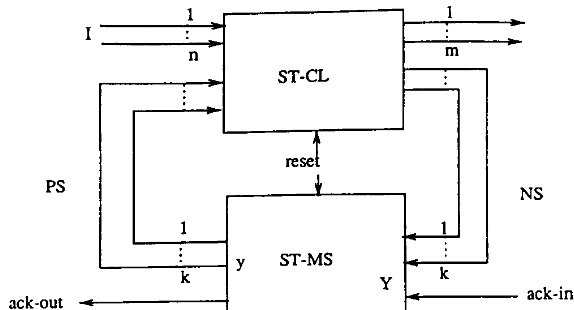

Fig. 3. The self-timed FSM. The self-timed CL implements the Boolean functions. The self-timed MS serves as the feedback register.

to those employed in [3]. In the following, we describe the general outline of the proof. The proof relies on the correct behavior of the underlying CL (as given in [3]) and MS (as discussed in Section II). The proof consists of the following two steps:

1) The FSM, as constructed, provides correct environments to both the CL and the MS. That is, the E_i environment constraints of [3] and Section II, respectively, are complied with in the order specified thereof.
2) The FSM, as constructed, satisfies the S_i constraints of Section III.

In addition, attention is given to initialization.

The proof is best based on the following description of FSM operation. Refer to Fig. 3 and to the sequential constraints of Section III. Assume we observe the FSM when all inputs to CL (I's and y's) are zero (undefined). Eventually, all CL outputs (O's and Y's) will also become zero. Consequently, the present state y will become defined, and "W" will become 1. This is a stable state of the FSM ($E0$), and it remains so until the environment chooses to respond to W = 1 by applying new inputs.

When all inputs I's become defined, and since the y's are already defined, all the CL's outputs (O's and Y's) become defined. Consequently, MS stores the next state Y, zeroes the present state y, and resets "W." This is also a stable state of the FSM, although it occurs in the middle of the "state transition." As long as the inputs remain defined, the FSM stays in this state, with the outputs O's defined and with the next state Y stored internally in the slave stage of the MS.

When the inputs are removed (by the environment), we return to the state we started with above. Now let us proceed to discuss the first step of the proof. To distinguish the different constraints, we employ a notation such as $E_i(CL)$, which is E_i of the sequential constraints of CL, and similarly $E_i(MS)$ and $E_i(FSM)$.

Theorem: If the sequential constraints of FSM are satisfied, then the sequential constraints of CL [3] are satisfied.

Proof: Assume $E1(CL)$ is granted. That is, the FSM's inputs I's and present state y are undefined. According to the description of operation of FSM above, $S1(CL)$ must occur before y can be defined, and before "W" turns 1 and allows the inputs to become defined. Hence, $E2(CL)$ follows $S1(CL)$. $E3(CL)$ occurs as a consequence of $S1(CL)$; all Y's and O's undefined cause all y's to become defined, and after "W" turns 1, the environment will turn all inputs defined. Now, the FSM's inputs I's and present state y are defined, $E3(CL)$. According to the description of operation of FSM above, $S3(CL)$ must occur before y can be undefined, and before "W" turns 0 and allows removal of the inputs. Hence, $E4(CL)$ follows $S3(CL)$. $E1(CL)$ will occur as a consequence of $S3(CL)$; all Y's and O's defined will cause all y's to become undefined, and after "W" turns 0, the environment will turn all inputs undefined, $E1(CL)$. Q.E.D.

Theorem: If the sequential constraints of FSM are satisfied, then the sequential constraints of MS (Section II) are satisfied.

Proof: Assume $E0(MS)$ is granted. That is, all Y's are undefined and all y's are defined. Hence, W = 1 and the FSM's environment may now turn the inputs I defined. When all inputs I's have become defined, (the y's are already defined), Y's turn defined and $E1(MS)$ holds. Eventually, $E2(MS)$ holds too. According to the description above, all y's turn undefined, thus removing some of the CL's inputs. Also, "W" turns 0, and signals the environment that it may remove the inputs I's. Only then are all the CL's input removed, and subsequently the Y's become undefined, i.e., $S2(MS)$ precedes $E3(MS)$. $E4(MS)$ (all Y's defined) follows $E3(MS)$. Similarly, all y's must turn defined, thus providing some inputs to the CL, and "W" turns 1, effecting subsequent definition of the inputs I's. Only then will the CL generate Y's. Hence, $S4(MS)$ precedes $E1(MS)$. Q.E.D.

As to the second part of the proof, we have shown by describing the operation of the FSM that, given the constraints on its environment $E_i(FSM)$, the $S_i(FSM)$ are satisfied. □

The discussion above assumes that the FSM has been initialized properly. This is not a straightforward assumption,

because if the FSM starts at some arbitrary state, there is no guarantee that it will ever reach a legal state, according to the cycle of activities described above. The most common solution employs a "sufficiently long" reset signal at initialization. Note that the memory elements in the FSM, i.e., those that need to be initialized, are all C-elements. As described in Section II, those elements may be constructed to include "preset" and "clear" inputs. We apply the reset signal, at initialization time, for some predetermined length of time, to the output C-elements of the MS and to all the C-elements in the CL. The reset signal is connected either to the clear or to the preset input of each C-element, as the case may be. The length of the reset signal is sufficiently long to cause all other C-elements, gates, and signal wires to settle at the desired initial state. Only after this step does the system start to behave as a self-timed system, as formalized in this paper. Incidentally, the reset circuitry can be simplified even further than proposed here.

Our design employs AND–OR "theoretical" gates. In practice we can use NAND–NOR gates instead, without significantly changing the circuits.

The FSM circuit produces an acknowledgment line ("W" in Fig. 1). This line is handled by the environment, and is used as a completion signal. We could employ an ack-in line, e.g., for "train-like" structures, where the ack line is fed from each FSM to the previous one. The ack-in line can simply be added as shown in Fig. 1. Whether to use the ack-in line or not to use it is very dependent on the implementation. The "train" structure can be efficient in some implementation, e.g., pipeline, where each subsystem waits for its successor to acknowledge receipt of the data (or spacer) and then continue its computation. Other implementations might prefer not to include the ack-in line, and let the external environment deal with all the acknowledgment lines in the system (e.g., collecting them in one big CE).

A fully worked-out example of a self-timed FSM, based on the traffic-light circuit of [9], can be found in [4].

V. Discussion

The scope and definition of self-timed circuits have been explained in the companion paper [3]. The limitations of ideal delay-insensitivity due to forks and assumptions of stability which are discussed in [3] apply to the design of self-timed FSM's as well.

Self-timed systems can be specified in a few different ways. One example is based on input–output sequences, that is, specification by means of "accepted language" [2], [10] or regular expression recognizers [1]. Other specifications are based on graph models such as the different types of Petri nets. CSP has also been used to specify self-timed systems [7], [8]. We use the state transition representation to specify ST-FSM; this representation is the one most often employed by engineers to design FSM's.

Self-timed systems can be employed for implementing control sections of systems where the control and the data path sections are separated. We do not enforce this separation. Indeed, our approach might lead toward dataflow-like designs.

Self-timed FSM's can be designed in different flavors. Our FSM resembles synchronous Moore and Mealy machines. Other proposals look more like *fundamental mode* [6] asynchronous sequential machines [2], [10]. Our FSM requires all inputs to be defined before a state transition can take place. In addition, in between state transitions all inputs must be removed ("spacer"). The FSM generates a completion signal once the transition is over. Outputs can depend on the state (Moore-type) or on the state and inputs (Mealy type), and both types can be combined in the same FSM. The inputs and outputs are ternary $(0, 1, U)$ and are implemented with double-rail codes. The implementation is simple.

The FSM is designed to be formally provable. It is scalable, in the sense that the design does not depend on contemporary technological constraints, such as the size of an equipotential region [12]. No internal delay elements are employed to guarantee correct operation, unlike [10]. All self-timed attributes are guaranteed by means of scalable logic, rather than any analog feature such as delay properties. As a result, the FSM design is systematic and yields to formal analysis and proof of correct behavior. However, the proof depends on certain stability assumptions [3].

Our goal is proof of correctness, hence our design is modular and hierarchical. For this purpose we trade off two other goals, namely space and speed. Alternative designs exist, that produce more efficient architectures, but either give up modularity and ease of proof, or relax some of the restrictions (e.g., make assumptions on relative delays).

Chu [2] develops a method for synthesizing delay-insensitive control circuits; he synthesizes the circuit from formal graph-theoretic specifications called Signal Transition Graphs. These graphs form a subclass of Petri nets, and correspond only to a subclass of FSM's. His method requires the separation of data path and control in the design, losing the nice similarity between self-timed and data-flow designs. The approach as presented does not include a complete algorithmic path from specification to implementation, and hence cannot be fully automated.

Molnar *et al.* [10] develop a synthesis method for delay-insensitive circuits based on Petri net specifications. The machine is characterized by input–output sequences and not by a state-table. Their method requires the addition of delay elements to produce the *ready* signal, and thus it is not entirely delay insensitive.

Martin [7], [8] compiles delay-insensitive circuits from CSP-like specifications. With this method, the modules are specified as sequential processes that can be refined into more detailed descriptions. Finally he replaces these descriptions with hardware templates that perform the actions required.

Anantharaman [1] proposes a method for synthesizing self-timed recognizers. Thus, his approach is restricted to single-input, single-output machines, specified by regular expressions.

Extensive research on the synthesis of delay-insensitive circuits is also reported in [11], [13], and [5].

VI. Conclusions

In this paper we describe the specification and implementation of a self-timed finite state machine. The specification of our FSM is given by a state table, similar to that of

synchronous machines. The circuit operates according to a sequence of events that replaces the role of the central clock in the synchronous FSM. The inputs and outputs of the circuit are double-rail (or ternary) and the circuit produces a completion signal. The correctness of the circuit can be proven formally, and thus self-timed FSM's can serve as "correct by construction" building blocks for system synthesis.

ACKNOWLEDGMENT

The authors wish to express their appreciation to the referees for their detailed and helpful comments that improved the quality of this paper.

REFERENCES

[1] T. S. Anantharaman, "A delay insensitive regular expression recognizer," *IEEE VLSI Tech. Bull.*, vol. 1, no. 2, Sept. 1986.
[2] T. A. Chu, "Synthesis of self-timed VLSI circuits from graph-theoretic specifications," Ph.D. dissetation, Massachusetts Institute of Technology, 1987.
[3] I. David, R. Ginosar, and M. Yoeli, "An efficient implementation of Boolean functions as self-timed circuits," *IEEE Trans. Comput.*, this issue, pp. 2–11.
[4] ——, "Implementing sequential machines as self-timed circuits," Tech. Rep. 692, Dep. Elec. Eng., Technion, Nov. 1988.
[5] J. C. Ebergen, "Translating programs into delay-insensitive circuits," Ph.D. dissertation, Eindhoven Univ. of Technology, 1987.
[6] Z. Kohavi, *Switching and Finite Automata Theory*, 2nd ed. New York: McGraw-Hill, 1978.
[7] A. J. Martin, "Compiling communicating processes into delay insensitive VLSI circuits," *Distributed Comput.*, vol. 1, no. 3, pp. 226–234, 1986.
[8] ——, "Programming in VLSI: From communicating processes to delay-insensitive circuits," Caltech-CS-TR-89-1, Dep. Comput. Sci., California Institute of Technology, 1989.
[9] C. Mead and L. Conway, *Introduction to VLSI Systems*. Reading, MA: Addison-Wesley, 1980
[10] C. E. Molnar, T. P. Fan, and F. U. Rosenberger, "Synthesis of delay-insensitive modules," in *Proc. 1985 Chapel Hill Conf. VLSI*, Chapel Hill, NC, May 15–27, 1985, pp. 67–86.
[11] M. Rem, "Concurrent computations and VLSI circuits," in *Control Flow and Data Flow; Concepts of Distributed Computing*, M. Broy, Ed. Berlin, Germany, Springer-Verlag, 1985, pp. 399–437.
[12] C. L. Seitz, , "System timing," in *Introduction to VLSI Systems*, C. Mead and L. Conway, Eds. Reading, MA: Addison-Wesley, 1980, pp. 218–262.
[13] J. L. A., Van de Snepscheut, *Trace Theory and VLSI Design*, LNCS 200, 1985.

Ilana David, for a photograph and biography, see this issue, p. 11.

Ran Ginosar (S'79–M'82), for a photograph and biography, see this issue, p. 11.

Michael Yoeli, for a photograph and biography, see this issue, p. 11.