Designing Self-Timed Devices Using the Finite Automaton Model

VICTOR I. VARSHAVSKY
VYACHESLAV B. MARAKHOVSKY
VADIM V. SMOLENSKY
University of Aizu

The authors suggest a procedure for designing a self-timed device defined by the finite automaton model. This procedure proves useful when designing these devices using the available synchronous behavior specifications. They illustrate the effectiveness of their procedure by applying it to the design of a stack memory and constant acknowledgement delay counter.

ASYNCHRONOUS SYSTEMS offer many advantages in terms of performance and power. Designing them, however, is essentially an art (see the Designers as artists box), and the quality of the final circuit implementation depends greatly on the designer's skill. Our research, therefore, defines a procedure that reliably accomplishes the routine work associated with designing one class of asynchronous systems, self-timed devices. We do not intend to take the designer out of the design process; rather, our procedure frees a designer to review more possibilities.

Our procedure synthesizes a self-timed device with external inputs from a finite Mealy automaton specification. We chose to use that specification and a two-register structure with master-slave flip-flops for two reasons:

- Representing behavior specifications in finite automata language is widespread and supported by many CAD systems. It also has a good theoretical and practical basis.
- As we will show, the Mealy automaton's self-timed realization with a two-register structure has a simple and evident solution. For the given examples, it possesses a complexity close to that of the synchronous realizations.

One reason preventing wide use of the conventional asynchronous approach is the necessity of anti-race coding, which causes major complications in an implementation. In this sense, self-timed realizations take the middle position between synchronous and asynchronous ones. The inputs and outputs of self-timed circuits usually have a two-phase behavior (code-spacer-code, ...)[1] and a four-phase interface with the environment (request-code-acknowledgement-spacer-request, ...). These characteristics of the inputs and outputs, in fact, organize synchronous behavior and therefore allow specification in synchronous finite automata language. However, the problem of obtaining correct circuit behavior (eliminating races, for example) stays outside the specification. Proper transitioning of the specification to the realization structure guarantees gate-delay insensitive behavior and provides the correct circuit behavior.

Such an approach is not a methodological novelty. This article illustrates its particular application to a situation

0740-7475/95/$04.00 © 1995 IEEE

that often arises when designing self-timed devices using the available synchronous-prototype behavior specification. For a summary of the research supporting this approach, see the Background box (next page).

Specification problems

We must first derive the device behavior specification. This process is informal and produces varying results. In addition, a specification of minimal complexity doesn't guarantee minimal implementation complexity. Changing input specifications, however, achieves a more significant improvement in circuit solution quality than perfecting formal synthesis procedures. (Of course, this doesn't mean that these procedures need not be perfected.)

We chose to present input specifications for formal methods as change diagrams that are extensions of signal graphs. Only a correct change diagram has a self-timed implementation. To achieve correctness, we might change the initial specification, for example, by inserting extra intermediate signals. Then we proceed from the correct change diagram to Muller's diagram, a special state graph. From the set of states in this graph, we obtain truth tables for signal functions and derive Muller's circuit. Reasonable insertion of intermediate signals into the initial specification often leads to a significant simplification of Muller's circuit. Researchers have studied inserting such signals automatically, but we do not consider their algorithms satisfactory.

To construct a change diagram specification of a device, we must at least know its input and output signals. For example, in the case of a control unit coordinating the interaction between asynchronous, concurrent processes, we know beforehand the signals initiating processes and the Acknowledge signals. For this case we can rather easily derive a change diagram description of the autonomous system (consisting of the control unit and the controlled processes) functioning where these signals represent asynchronous processes. In such a system, inserting intermediate signals to simplify realization of the control unit is the problem.

More difficult specification problems arise if 1) device input and output signals are not defined beforehand, and 2) a deterministic model cannot describe environmental behavior. In the first case, the designer can design a preliminary device structure as a basis for choosing signals necessary for the specification. The second case requires either inserting a vertex of free choice into the model and modernizing the formal method, or "unwinding" the specification (linearization).

Designers as artists

Despite successes of theory and CAD developments, logic design is still an art and will apparently remain so for a long time. Many designers and researchers understand this fact perfectly, as confirmed by the frequent appearance of brilliant, new circuit solutions in various articles and patents. The authors of one book state

> Different types of support are needed during each step or phase. For example, during formulation and generation, designers tend to be "artist," while during evaluation and selection, designers are "analyst." The artist versus analyst distinction has clear and important implications for how the structure and flexibility of the support should vary across design phases.[1]

Designers usually display their skills in a fortuitous choice of a basic structure within which they derive the device behavior specification. It is difficult, however, to optimize a specification, and vague optimization criteria often compound the difficulties.

Unfortunately, no regular way exists to connect the initial structure (within which designers seek a solution) directly to the expected result. Experience shows that different devices are easier to design using certain structures. Currently, designers merely search among the various structures and specifications. Our goal is to develop a design procedure that starts with Mealy's finite-automaton specification.

References

1. *System Design: Behavioral Perspectives on Design, Tools, and Organization*, W.B. Rouse and K.R. Boff, eds., North Holland, Amsterdam, 1987, p. 8.

Design procedure

Our procedure, then, intends to obtain a correct change diagram specification from a finite automaton model. Although self-timed devices are asynchronous automata, we will not use asynchronous automaton models to define them. This is because these models solve problems like antirace coding of internal states and hazard-free implementation of logic functions. A self-timed device's main design problem is fixation of transient process completion moments in its circuit. The circuit certainly must be free of races and hazards, too, but fighting them is an attendant problem. Global methods handle races and hazards. They also often lead to better circuit solutions than that of special

Background

A nonautonomous automaton must interact with an environment which forms admissible input sequences for it and senses its processing results. For a system comprising a finite automaton and its environment, three structural models reflect various approaches to timing organization: synchronous, asynchronous, and matched. We are interested in the matched model in which environment forms an automaton work step, and the automaton forms a step of the environment work.

The matched model uses a handshake method as its basic mechanism. In general, a request denotes change of the input state, and an acknowledgement denotes change of the output state. Since we use code for automaton inputs and outputs, it would be more correct to say "change of a class of input or output states."

We define a self-timed automaton as a matched automaton in which a change in an output-state class finishes the transition caused by a change in an input-state class. This occurs irrespective of delays in the elements constructing the automaton.

This definition is not formal and emphasizes only that a self-timed automaton must correctly perform the automaton conversion under any ratio of delays of its elements. This requirement is feasible when using special coding systems and restrictions on the characteristic of delays in elements and wires.

The following Muller's hypothesis of delay characteristics conforms well to practice: 1) delays can be both inertial and pure; 2) delays in elements and pieces of wires from an element output up to a fork can be of any finite value; and 3) wires after a fork have a skew in delay values not more than the minimum delay of an element. In general, designers use self-synchronized code systems to code the input, output, and internal states of an automaton.

Our earlier work proved the possibility of designing a self-timed implementation of an arbitrary finite automaton consisting of a combinational circuit and memory elements. We also developed methods to synthesize self-timed automata from electric-potential elements. To do this, we use methods and procedures designing self-timed realizations of Boolean function systems, memory elements, and the circuits signaling transition processes completion in those elements.

To create formal methods of self-timed circuit synthesis, we need formal models. These models must reflect possible work concurrency and interaction asynchronism between different parts of the device. In this case, unfortunately, a finite automaton model representing a sequential machine is useless.

Known formal methods of self-timed device synthesis use dynamic models for specifying parallel asynchronous processes in circuits. These models include Muller's diagrams,[1] signal graphs, and change diagrams.[2] Synthesis methods based on such models work well for designing autonomous devices. When synthesizing devices with external inputs and outputs, the specification-simulating environment behavior must complement the general device behavior specification. This insertion usually requires "specification linearization,"[2] a multiple unwinding of the general specification that significantly complicates both the specification procedure and synthesis. Also, some research derives the general specification of the "device-environment" system using vertices of free choice[3] rather than linearization.

On the other hand, the well-developed and widely used language of finite automata allows simple determination of the required device-environment interaction behavior. The idea of joining the advantages of carefully studied finite-automaton models with methods of self-timed structure specification and synthesis, while desirable, is not original. We began investigating self-timed structures exactly from such models. Several studies using finite automaton models to design self-timed devices exist.[3,4]

References

1. D.E. Muller and W.S. Bartky, "A Theory of Asynchronous Circuits," Reports 75 and 78, Digital Computer Lab., University of Illinois, 1956, 1957.
2. M.A. Kishinevsky et al., *Concurrent Hardware: The Theory and Practice of Self-Timed Design*, J. Wiley and Sons, London, 1993.
3. T.A. Chu, "Synthesis of Hazard-Free Control Circuits from Asynchronous Finite State Machine Specifications," *Proc. TAU—ACM Int'l Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*, Princeton University, Princeton, N.J., 1992, p. 28.
4. T.A. Chu, "An Efficient Critical Race-Free State Assignment Technique for Asynchronous Finite State Machines," *Proc. 30th Design Automation Conf.*, Assn. of Computing Machinery, New York, 1993, pp. 2-5.

methods developed within the theory of asynchronous automata.

We use the classic finite Mealy automaton model for designing a self-timed device. The transition graph, shown in Figure 1, presents such an automaton.

In this graph, S_i and S_j belong to the set of internal states; X_k and Y_l belong to the sets of input and output characters.

The pair X_k/Y_l marks each arc leading from one state to another. The automaton passes from state S_i to state S_j under the influence of input X_k producing output Y_l.

Let us consider only those automata in which each internal state is attainable from any other state, that is, those with connected transition graphs. This restriction is not excessive, because, first, connected automata are of the greatest interest. Second, any unconnected graph can always be converted to a connected one by using dummy input and output characters.

Our procedure for designing a self-timed device consists of the following steps:

1. choice of a standard self-timed realization for a finite automaton
2. reduction of an automaton transition graph to a simple cycle graph
3. construction of a change diagram device specification
4. application of formal methods to the change diagram to obtain Muller's circuit for output and memory element excitation signals

Standard automaton realization. Step 1 defines the automaton structural scheme; the rules governing its interaction with the environment; coding input, output, and internal-state characters; and the memory element structure. Only logic functions of the allotted signal stay undefined. Now we can define the partial order of signal change for the signal sets representing the automaton's inputs and outputs as well as the memory element signals. However, the environment's nondeterministic behavior (it chooses the next input set in an unknown way) is a problem.

Two studies[1,2] propose canonical approaches that allow us to simplify the synthesis procedure. Using them, we obtain standard realization circuits. The memory element determines the type of standard realization. Standard realizations use irredundant coding of automaton internal states, unlike asynchronous automata, which use antirace coding. A standard realization contains a combinational circuit, parallel register, and perhaps, an indicator of transient process completion moments. D flip-flops with two-rail inputs or T flip-flops form the register. In this article, we use a memory element of two master-slave RS flip-flops.

Special design methods, as well as proper information coding, provide invariability of circuit behavior from element time parameters. Such code systems are self-synchronized. For self-synchronized codes, set B appearing at the output indicates the completion of a transition from set A to set B. Self-synchronized codes are a universal and unique tool for fighting functional hazards.

Asynchronous automata theory usually uses neighbor or quasi-neighbor coding methods. Such coding systems can be treated as single-phase self-synchronized codes that are overly redundant. They allow direct transition from one character to another.

Those self-synchronized codes where every working character alternates with an empty one (spacer) are more convenient. Note that using such sequences removes the restriction peculiar to asynchronous automata that no character can follow itself. A working input character initiates the active phase of the device work; the spacer initiates the passive phase. The self-synchronized code of such sequences are two-phase.

The most commonly used two-phase, self-synchronized codes are equal-weight codes and codes having an identifier (Berger's codes). These are the basis of all other self-synchronized code systems. We prefer equal-weight codes consisting of sets with a fixed number of ones. An example is two-rail code that represents each information bit by two

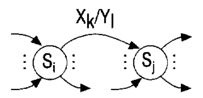


***Figure 1.** Mealy automaton transition graph.*

binary variables representing the active-phase contrary values. Spacers are sets consisting either of all zeroes or all ones. Optimum equal-weight code (Sperner's code) has the least redundancy. Its sets of length m contain $m/2$ ones.

In standard realizations, we usually use two-phase self-synchronized code systems for input and output states. That system defines the automaton-environment interaction. The automaton replies to a working set with a working set and to a spacer with a spacer; the environment replies to a spacer with a working set and to a working set with a spacer. It is also important to choose rules governing the interaction between different parts of the automaton structural diagram. We discuss this problem later.

Reducing the automaton graph. In step 2, we unwind the automaton transition graph into a simple cycle to change free choice of the next input character to a deterministic choice. The unwound graph defines a sequence of input characters that causes all the possible transitions in the automaton and is a loop that must pass every arc at least once. This guarantees the definition of all possible transitions. In such an unwinding, some states can meet repeatedly.

We can reduce a connected, oriented graph to a simple cycle in more than one way. We may wish to find its optimal unwinding; that is, the one containing a minimum number of vertices. The following is a possible algorithm for doing so:

1. compile the set of possible simple cycles from the automata graph

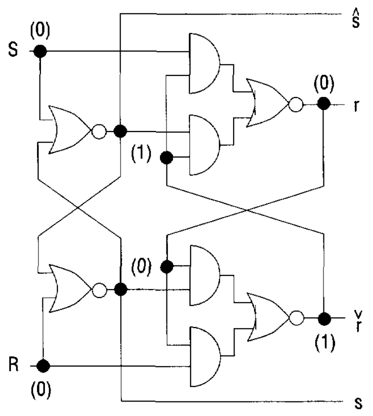


Figure 2. Memory element based on two flip-flops.

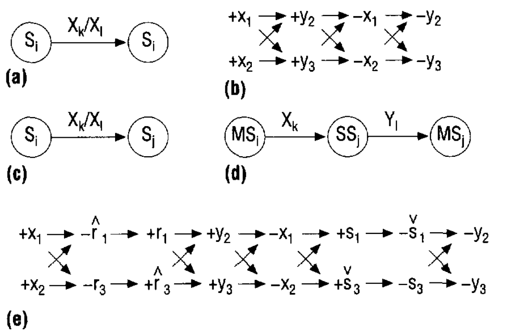


Figure 3. Possible transitions in the unwound automaton graph and their representation by signal graphs: no state change (a), partial order of signal change (b), state change (c), successive subtransitions (d), and a signal change order (e).

2. find all possible subsets of this set that cover all the arcs of the graph
3. choose the subset with the minimum number of cycle vertices

Any coverage of the graph is adequate for our purposes because all coverages define the same automaton; the optimal coverage simply reduces the subsequent work of the designer.

A designer next reduces the automaton graph to a simple cycle graph by

1. removing any two cycles containing at least one common vertex from the coverage
2. breaking both cycles by disconnecting the arc between the chosen common vertex and the destination, then connecting the broken cycles to form a new cycle and returning it to the coverage
3. repeating this process until one cycle remains in the coverage

This algorithm converges if the initial automaton graph is connected.

Constructing change diagram specifications. After unwinding the transition graph, we construct a linearized change diagram specification of the device on the allotted signal set. The standard realization type determines the way we construct it.

Two flip-flops with heteropolar control represent every variable coding internal state of the automaton. Figure 2 shows a possible circuit of such a memory element. It contains four AND-NOR gates with output signals s,ŝ and r,ř. The same characters indicate both the gate outputs and signals. The gates form two simple RS (s,ŝ) and $\overline{RS}$ (r,ř) flip-flops working as master-slave. A RS flip-flop changes its state through transit state (0,0) and a $\overline{RS}$ flip-flop changes its state through (1,1). The signs ^ and ˇ mark the inverted outputs of the flip-flops.

Set signals S and R are the external inputs of both flip-flops. Figure 2 shows the memory element initial state that corresponds to keeping the value of zero. The value one is written into a memory element in two phases. In the first phase, signal S = 1 switches the left flip-flop; in the second phase, when S = 0, information moves from the left flip-flop to the right. Double change of signal R writes zero in the same way.

To construct the standard realization of an automaton we must code input, output, and internal-state characters. Let two-phase, equal-weight, self-synchronizing codes represent input and output characters. This establishes the rules of interaction between the automaton and the environment. We use binary code to represent the internal states.

After coding, we must define the rules of interaction between different parts of the automaton structural scheme. After that we can derive the change diagram specification. Let us accept the following rules:

1. Every transition executes in two phases. The first phase starts with a set of self-synchronizing code at the automaton input and finishes when a set of code appears at the output. The second phase starts with a spacer at the input and finishes when a spacer appears at the output.
2. If the transition does not cause a change of state, then the input signals are the immediate cause of output signal changes. Figure 3a shows such a transition.
3. We break an automaton transition from one state to another (Figure 3c) into two successive subtransitions.

Figure 3d shows the subtransitions. Automaton states S_i and S_j correspond to states MS_i and MS_j of its memory element master flip-flop. MS_i changes to MS_j through intermediate state SS_j represented by the memory element slave flip-flop. A self-synchronizing code set at the automaton input initiates the first transition phase and causes switching of the necessary number of slave flip-flops. This leads to the appearance of a self-synchronizing code set at the output indicating phase completion. In the second phase, when the spacer appears at the input, the spacer writes the slave flip-flop states into the master flip-flops. After this, the spacer appears at the outputs of the automaton, indicating second phase completion.

Let us code the input and output characters of the automaton so that variables x_1 and x_2 of set X_k, and variables y_2 and y_3 of set Y_l have the value one, and the spacers include only zeroes. Then a transition like the one shown in Figure 3a corresponds to the partial order of signal change in Figure 3b.

Assume the codes of states S_i and S_j differ in variables s_1 and s_3 and in state S_i ($s_1 = 0$ and $s_3 = 1$). Then by coding characters X_k and Y_l in the same way we can easily represent the transition of Figure 3c by the signal change order in Figure 3e. Note that this graph has no flip-flop excitement signals. During synthesis, we will build excitement functions into the flip-flop gates. Such an approach allows the synthesis system to derive the necessary types of flip-flops (RS, D, or T).

We thus construct the signal change orders for every transition of the automaton graph unwound to a simple cycle. After defining the initial marking, we obtain the signal graph specification for the device.

We emphasize that the specifications obtained from different unwindings of an automaton graph, when processed by the synthesis procedure, must lead to the same result. We can strictly prove this statement.

Self-timed stack memory

As an example, we first applied our procedure to the design of a self-timed stack memory. Several approaches exist for designing this kind of memory; we divide them into two basic classes: register structures and memory-based stack. Studies of self-timed stack design usually consider register structures.[3] We will consider the second approach.

In a usual CMOS static-memory array with two-rail representation of the data path, sufficiently simple tools indicate read-operation completion. The main problem is indicating write-operation completion. We solved this problem[4] by breaking the write process into two phases: reading information and rewriting it. Memory detects rewrite completion by checking whether the code being written coincides with the code in the data path of the read. The details of self-timed memory organization and designing the control circuits are outside this article's scope.

Figure 4 presents the stack structure. It contains a self-timed memory array, stack pointer, and control unit. Signals R and W initiate read and write operations. They first enter the self-timed memory array and control unit to determine the stack pointer work mode. Ack is the signal that acknowledges operation completion to the environment. Adr is the set of address signals coming to the memory array from the stack pointer. Adr initiates the memory work, while R and W choose the mode. C is the set of control signals coming to the stack pointer from the control unit.

The signal graph in Figure 5 describes the rules governing stack block interactions. We indicate active and passive signal states by + and –. Extra signal O unites W and R making them indistinguishable.

Our goal is to design the control unit. To do this, we must consider the stack pointer circuit, which produces memory block addresses.

Stack pointer. The stack pointer logic circuit should be simple because its complexity grows linearly with increasing stack size. We achieve this simplicity by increasing the number of external control signals and, hence, complicating the control circuit.

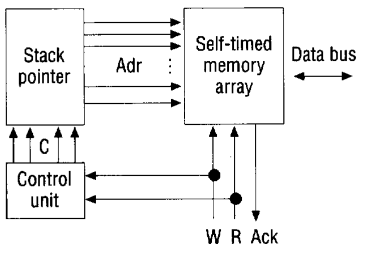


Figure 4. Stack structure.

$$\begin{tikzcd}
+O \arrow[r] & +C \arrow[r] & +Adr \arrow[r] & +Ack \arrow[d] \\
-Ack \arrow[u, "\bullet"] & -Adr \arrow[l] & -C \arrow[l] & -O \arrow[l]
\end{tikzcd}$$

Figure 5. Rules governing stack block interactions.

Figure 6, next page, shows such a stack pointer circuit. It is based on a multistable flip-flop with multiphase control. Such a flip-flop can come to stable states with violations of strict alternation of low and high levels at the gate outputs.

For example, in the sequence, ..., 010100101 ..., neighbor state pair 00 is a pointer. Compulsorily drawing the left gate of the pair 00 into state 1 (00→10) shifts the pointer to the left. To shift the pointer to the right, we draw the right gate of the pair 00 into the state 1 (00→01). Thus, there are two control signals necessary for shift control, for example, D and U. However, if we change any of these signals once, the pointer will move to one of the sides without stopping.

To prevent a through shift of the pointer, we double the number of the control signals and divide them into even (D_e, U_e) and odd (D_o, U_o), as in the circuit of Figure 6. The circuit keeps the pointer when all the control signals equal 1. If the pointer is on an even position (corresponding to the two middle gates of the circuit) then $D_e = 0$ shifts

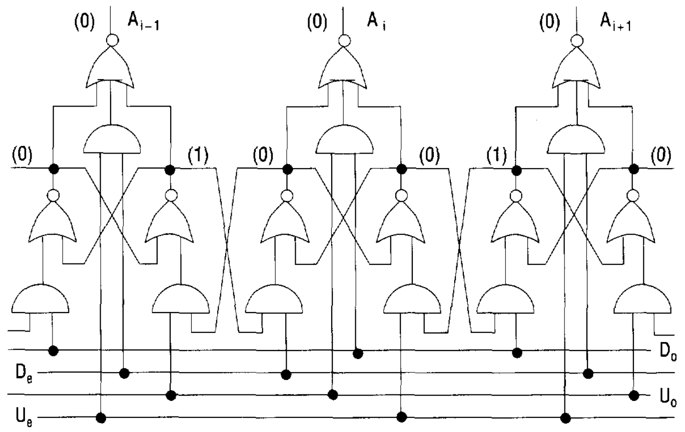


Figure 6. Stack pointer.

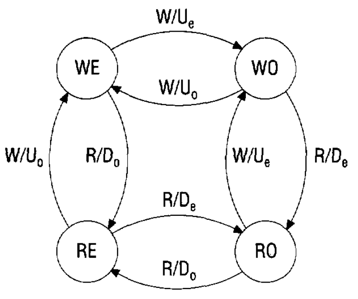


Figure 7. Control unit automaton graph.

it to the left odd position and $U_e = 0$ to the right. The odd pointer positions in Figure 6 correspond to the two left and the two right gates. From these positions, $D_o = 0$ shifts the pointer to the left, and $U_o = 0$ shifts it to the right.

The pointer position corresponds to the memory array address line. Signal $A_i = 1$ chooses the ith address line. The stack pointer determines values of A_i according to the pointer position and control signal values.

Control unit. This unit produces the stack pointer control signals using input signals W and R. The definition of this automaton behavior should describe, in particular, the interaction between the stack and the environment. Stack pointer behavior determines stack behavior. We represent it as follows:

- If a write operation precedes a write, $U_e = 0$ shifts the pointer one position up from an even position. $U_o = 0$ shifts it one position up from an odd position. After this, the stack pointer produces the address signal.
- If a write operation precedes a read, pointer position does not change, and $U_o = 0$ or $D_o = 0$ produces the address for an even position; $U_e = 0$ or $D_e = 0$ for an odd position.
- If a read operation precedes a read, $D_e = 0$ shifts the pointer one position down from an even position. $D_o = 0$ shifts it one position down from an odd position. The stack pointer then produces the address signal.
- If a read operation precedes a write, the pointer's position does not change, and $U_o = 0$ or $D_o = 0$ produces the address for an even position; $U_e = 0$ or $D_e = 0$ for an odd position.
- $U_e = U_o = D_e = D_o = 1$ will reset the address signal.

Using this description, we can easily represent the control unit by a Mealy automaton with the transition graph of Figure 7. The automaton states correspond to the following situations:

WE—occurrence of a write by an even address
WO—occurrence of a write by an odd address
RE—occurrence of a read by an even address
RO—occurrence of a read by an odd address

To code four states, coding variables S_1 and S_2 suffice. Let variable S_1 take values from set {e,o} and variable S_2 take values from set {r,w}. Then, clearly, the coding of the automaton states is neighbor since only one variable value changes in every transition. Such coding simplifies the automaton realization.

We set the output signals, shown at the arcs of the graph in Figure 7, so that every variable that codes internal states breaks its set into two nonoverlapping subsets. Choosing such output signals must simplify its logic functions.

For synchronous automaton realization, the universal antirace method uses a two-register memory that divides the registers' work in time. We find a similar approach useful for self-timed realization. For example, a pattern of self-synchronizing code appearing at the inputs triggers a write to the first register. A spacer at the input causes information to move from the first register to the second. In some cases, the second register can contain a smaller number of simple flip-flops than the first. When the information moves, it must be compressed. We use this case in our example.

Let variable S_1 correspond to RS flip-flop Y_e, Y_o and variable S_2 to RS flip-flop Y_r, Y_w. These two flip-flops form the first register. Since in every transition only one of these flip-flops changes its state, the second register can consist of one flip-flop. Let it be flip-flop (Z_1, Z_2).

Such automaton memory organization, together with coding input and output characters using two-phase self-synchronizing codes, provides monotonic representations for all signal logic functions. In our case, the inputs and outputs are already coded, and we can easily check that these codes are two-phase and self-timed. We have thus designed the standard realization of the automaton defined by the transition graph of Figure 7.

This graph reduces to a simple cycle graph rather easily. Two simple cycles, obtained by passing all the graph vertices clockwise and counter-clockwise, cover all the graph arcs. We construct an unwinding of the graph from these two cycles.

We derive the linearized specification, represented by the signal graph in Figure 8, by unwinding the graph of the automaton standard realization. Using this specification and formal synthesis methods, we obtain the following Muller's circuit:

$$
\begin{aligned}
Y_e &= \overline{Y_o \vee WZ_2 \vee RZ_1} \\
Y_r &= \overline{Y_w \vee W} \\
Y_o &= \overline{Y_e \vee WZ_1 \vee RZ_2} \\
Y_w &= \overline{Y_r \vee R} \\
Z_1 &= \overline{Z_2R \vee Z_2W \vee Z_2U_eD_o} \\
Z_2 &= \overline{Z_1R \vee Z_1W \vee Z_1D_eU_o} \\
U_e &= \overline{Y_oZ_2Y_w} \qquad D_e = \overline{Y_oZ_1Y_r} \\
U_o &= \overline{Y_eZ_1Y_w} \qquad D_o = \overline{Y_eZ_2Y_r}
\end{aligned}
$$

Considering these derivations as logic functions of AND-NOR gates, we can draw the logic circuit of the control automaton.

Modulo-2^n counter with a constant response time

Our second application example has many solutions.[5-7] Each solution uses two basic elements of structural scheme realization:

$$\begin{tikzcd}
+W \arrow[r] & -y_1 \arrow[r] & +y_2 \arrow[r] & -U_e \arrow[r] & -W \arrow[r] & +z_1 \arrow[r] & -z_2 \arrow[r] & +U_e \arrow[d] \\
+D_e \arrow[d] & -z_1 \arrow[l] & +z_2 \arrow[l] & -R \arrow[l] & -D_e \arrow[l] & -y_4 \arrow[l] & -y_3 \arrow[l] & +R \arrow[l] \\
+R \arrow[r] & -y_2 \arrow[r] & +y_1 \arrow[r] & -D_o \arrow[r] & -R \arrow[r] & +z_1 \arrow[r] & -z_2 \arrow[r] & +D_o \arrow[d] \\
+U_o \arrow[d] & -z_1 \arrow[l] & +z_2 \arrow[l] & -W \arrow[l] & -U_o \arrow[l] & +y_3 \arrow[l] & -y_4 \arrow[l] & +W \arrow[l] \\
+R \arrow[r] & -y_3 \arrow[r] & +y_4 \arrow[r] & -D_o \arrow[r] & -R \arrow[r] & +z_1 \arrow[r] & -z_2 \arrow[r] & +D_o \arrow[d] \\
+D_e \arrow[d] & -z_1 \arrow[l] & +z_2 \arrow[l] & -R \arrow[l] & -D_e \arrow[l] & +y_2 \arrow[l] & -y_1 \arrow[l] & +R \arrow[l] \\
+W \arrow[r] & -y_4 \arrow[r] & +y_3 \arrow[r] & -U_e \arrow[r] & -W \arrow[r] & +z_1 \arrow[r] & -z_2 \arrow[r] & +U_e \arrow[d] \\
+U_o \arrow[uuuuuuu, bend left] & -z_1 \arrow[l] & +z_2 \arrow[l] & -W \arrow[l] & -U_o \arrow[l] & +y_1 \arrow[l] & -y_2 \arrow[l] & +W \arrow[l]
\end{tikzcd}$$

Figure 8. *Control automaton signal graph.*

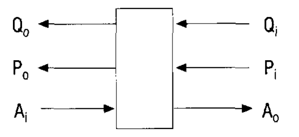


Figure 9. *Modulo-2^n counter stage.*

- An arrangement of counter stages in a pipeline guaranteeing that the response time is independent of the number of stages
- A counter contains two interconnected asynchronous pipelines. Counting signals propagate in one; carry and overflow signals propagate in the other, but move in opposite directions.

Let us construct such a counter using these principles and our automaton approach. In a modulo-2^n counter, n is the number of stages. Let the ith counter stage be the black box shown in Figure 9. It receives counting signals from the (i–1)th stage at input A_i and replies to them at output Q_o if there is no overflow, or at output P_o if overflow occurs. The ith stage passes the counting signals to the (i+1)th stage through output A_o; Acknowledge signals respond to these signals at inputs Q_i or P_i, depending on the presence of overflow. The modulo-2^n counter consists of n such stages connected in series. The least significant stage interacts with the environment.

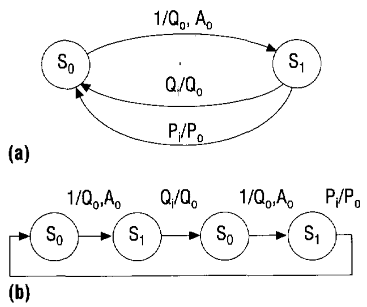


Figure 10. *Transition graph of the modulo-2^n counter stage **(a)** and unwinding of the graph **(b)**.*

Free inputs of the most significant stage meet the following boundary conditions: input P_i connects to output A_o and constant zero feeds input Q_i.

A finite automaton with the graph shown in Figure 10a represents a counter stage. The automaton has internal states S_0 and S_1 that correspond to keeping the information bit values 0 and 1. Signal $A_i = 1$, which is the analog of clock signal for the automaton, initiates every transition. Therefore, this signal does not appear in the transition graph. The automaton passes from state S_0 to state S_1 on any condition; on the corresponding transition arc the condition is marked 1 and the output signals are Q_o = 1 and A_o = 1. The automaton allows the transition from S_1 to S_0 only when it receives input signal

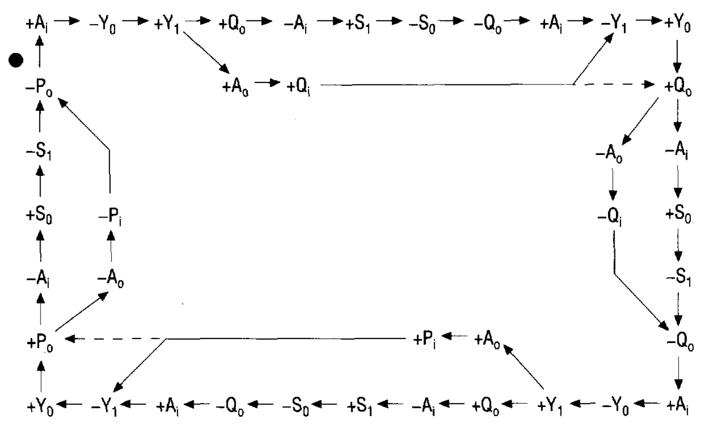


Figure 11. *Signal graph specification of the counter stage.*

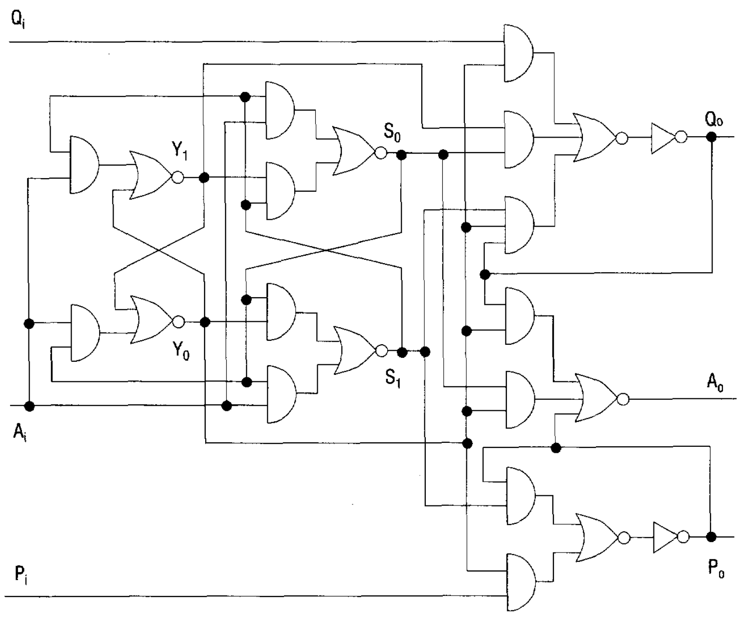


Figure 12. *Logic circuit of the counter stage.*

$Q_i = 1$ or $P_i = 1$. Two arcs depict this transition since different output signals can be produced during it: $Q_o = 1$ when $Q_i = 1$, and $P_o = 1$ when $P_i = 1$.

One coding variable S suffices to code two internal states. Let two simple flip-flops correspond to it. They are a slave flip-flop (Y_0, Y_1) with transit state 00 and a master flip-flop (S_0, S_1) with transit state 11. The inputs and the outputs of the automaton are already coded. We can easily check that for all transitions coming from any internal state the codes of the inputs and outputs are self-timed and two-phase. Thus, we have constructed a standard automaton realization.

Let us unwind the automaton graph to a simple cycle (Figure 10b). Using this unwinding and the accepted standard automaton realization, we derive the signal graph specification, shown in Figure 11.

From this specification, formal methods produce the following Muller's circuit:

$$\begin{aligned}
Y_0 &= \overline{Y_1 \vee A_i S_0} \\
Y_1 &= \overline{Y_0 \vee S_1 A_i P_i \vee S_1 A_i Q_i} \\
S_0 &= \overline{S_1 Y_1 \vee S_1 A_i} \\
S_1 &= \overline{S_0 Y_0 \vee S_0 A_i} \\
Q_o &= Y_0 Q_i \vee Y_1 S_0 \vee Q_o Y_0 S_1 \\
P_o &= Y_0 P_i \vee P_o S_1 \\
A_o &= Y_1 \vee A_o \overline{Q}_o \overline{P}_o
\end{aligned}$$

Note that in this circuit the functions of signals Q_o, P_o, and A_o are self-depending, that is, they must be implemented as flip-flops made from a gate and an inverter. Signals $\overline{Q}_o$ and $\overline{P}_o$ in function A_o come from the gate outputs of the corresponding flip-flops.

We can improve this solution by inserting changes into the signal graph specification. For example, the modified specification derived by removing two arrows, shown as dashed lines in Figure 11, provides simpler functions for Y_1 and A_o

$$\begin{aligned}
Y_1 &= \overline{Y_0 \vee A_i S_1} \\
A_o &= \overline{P_o \vee Y_0 S_0 \vee Y_0 Q_o}
\end{aligned}$$

Other signal functions do not change. In this way, we can easily draw the counter stage logic circuit (Figure 12) from the Muller's circuit.

We analyzed the performance of the counter circuit, measuring response time by the number of gates switched. That number equaled four for every change of the counting signal and did not depend on the number of stages. Our solution, then, does not exceed those already known in complexity.[5-7]

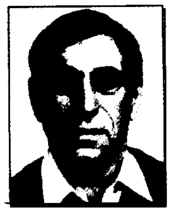
LIKE ANY DESIGN PROCEDURE, ours contains both formal and informal steps. Choosing the specification language, behavior specification, and basic structure is informal. Designers use their experience to make these choices. These choices, in turn, determine the quality of the solution.

Transitioning the specification within a basic structure to a change diagram is a formal step. It allows us to obtain an self-timed implementation using well-known formal methods.[2] The facilities used for formal synthesis determine solution quality.

It is important to introduce the behavior of the basic-structure gates into the change diagram. Starting our design procedure at the gate level increases the quality of the solution. This article's limited length does not allow us to describe the details of this process.

References

1. A. Astanovsky et al., *Aperiodical Automata*, V. Varshavsky, ed., Nauka, Moscow, 1976 (in Russian).
2. V.I. Varshavsky et al., *Self-Timed Control of Concurrent Processes*, V. Varshavsky, ed., Kluwer Academic Publisher, Boston, 1990. (Russian, 1986).
3. M.B. Josephs and J.T. Udding, "The Design of a Delay-Insensitive Stack," in *Designing Correct Circuits, Workshops in Computing*, G. Jones and S. Sheeran, eds., Springer-Verlag, New York, 1990, pp. 132-152.
4. V.I. Varshavsky et al., "Memory Device from MOS Transistors," USSR patent, Certificate No. 1365129, ICI G 11c 11/40, *The Inventions Bulletin*, No. 1, 1988.
5. J.C. Ebergen and M.G. Peeters, "Modulo-N Counters: Design and Analysis of Delay-Insensitive Circuits," *Proc. Second Workshop on Designing Correct Circuits*, North Holland, Amsterdam, 1992, pp. 27-46.
6. C.D. Nielsen, "Performance Aspects of Delay-Insensitive Design," PhD thesis, Technical U. of Denmark, Lingby, Denmark, 1994.
7. K. van Berkel, "VLSI Programming of a Modulo-N Counter with Constant Response Time and Constant Power," *Asynchronous Design Methodologies*, IFIP, Vol. A-28, S. Furber and M. Edwards, eds., Elsevier Science Publishers, New York, 1993, pp. 1-11.



Victor I. Varshavsky is a professor and head of the Computer Logic Design Laboratory at the University of Aizu, Japan. His research interests are in the area of operation research, automata theory, logic design, and computer engineering. Varshavsky received the DEng degree in precise mechanics from the Leningrad Institute of Precise Mechanics and Optics. He also holds a PhD degree from the Leningrad Institute of Aviation Instrumentation and a DrEng from the Institute of Control Problems, Russian Academy of Science; both degrees are in engineering cybernetics.

Vyacheslav B. Marakhovsky is also a professor at the University of Aizu and is in charge of the Computer Networks Laboratory. His research interests include automata theory, logic design, and computer engineering. Marakhovsky holds the DEng in electrical engineering from the Leningrad Polytechnical Institute, a PhD in engineering cybernetics from the Central Institute of Economics and Mathematics, Russian Academy of Science, and DrEng in computer science from Leningrad Electrotechnical University. He is a member of the IEEE.

Vadim Smolensky works at the University of Aizu as a research associate, where he participates in the Self-Timing and Event-Driven Systems research project. Smolensky received the DEng in computer science from Leningrad Electrical Engineering Institute.

Address questions or comments about this article to Victor I. Varshavsky, University of Aizu, Tsuruga, Ikki-machi, Aizu-Wakamatsu City, Fukushima, 956 Japan; victor@u-aizu.ac.jp.